Optimisation d'un dispositif hyper-sustentateur par algorithmes génétiques

Mathieu Renversade

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Optimisation d'un dispositif hyper-sustentateur par Algorithmes Génétiques

Mathieu Renversade

N° 4029

Octobre 2000

THÈME 4

ISSN 0249-6399 ISRN INRIA/RR--4029--FR

INRIA
SOPHIA ANTIPOLIS

Optimisation d'un dispositif hyper-sustentateur par Algorithmes Génétiques

Mathieu Renversade

Thème 4 — Simulation et optimisation
de systèmes complexes
Projet SINUS

Rapport de recherche n° 4029 — Octobre 2000 — 111 pages

Abstract: This report presents the optimization of the aerodynamic performances of a high-lift device by genetic algorithms. A compressible flow solver based on the Euler equations and designed on unstructured triangular meshes is coupled with an evolutive-type optimization algorithm, in order to maximize the lift. A program written in Fortran 77 is first tested, then a software called EASEA, is used to generate a genetic optimization code in C++, sequential first, then in a parallel version based on MPI process group features. With this C++ code, several genetic operators are tested, as well as other strategies such as a Nash game. Performances and results are finally compared.

Key-words: Optimization, Genetic Algorithms, Euler Equations, EASEA, High-lift, Nash, C++

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, B.P. 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : 04 92 38 77 77 - International : +33 4 92 38 77 77 — Fax : 04 92 38 77 65 - International : +33 4 92 38 77 65

Optimisation d'un dispositif hyper-sustentateur par Algorithmes Génétiques

Résumé : Ce rapport présente l'optimisation des performances aérodynamiques d'un dispositif hyper-sustentateur par algorithmes génétiques. Un solveur de flot compressible fondé sur les équations d'Euler implémenté sur un maillage triangulaire non structuré est couplé à un algorithme d'optimisation de type évolutif, afin de maximiser la portance du dispositif. Un programme écrit en Fortran 77 est en premier lieu testé, puis un logiciel, nommé EASEA, est utilisé afin de générer un code d'optimisation génétique en C++, d'abord en séquentiel, puis dans une version parallèle s'appuyant sur les primitives MPI. Avec ce code C++, plusieurs opérateurs génétiques sont testés, ainsi que d'autres stratégies, telles celle de Nash. Les performances et résultats des différentes méthodes sont ensuite comparés.

Mots-clés : Optimisation, Algorithmes Génétiques, Equations d'Euler, EASEA, hyper-sustentateur, Nash, C++

Table des matières

Liste des tableaux

Table des figures

Introduction

Ce rapport présente le travail effectué au cours d'un stage de six mois à l'INRIA (Institut National de Recherche en Informatique et en Automatique), au sein du projet SINUS (SImulation NUmérique pour les Sciences de l'ingénieur), d'Avril à Septembre 2000. L'INRIA, organisme de recherche public, intervient dans des domaines très variés, tous liés aux technologies de l'information. Le projet SINUS s'attache à faire progresser les méthodes de modélisation numérique, depuis l'analyse des modèles mathématiques ou physiques jusqu'à la mise en œuvre sur ordinateur des algorithmes de résolution. Le domaine d'application actuel est celui des écoulements compressibles turbulents ou réactifs, en relation avec des partenaires industriels (aéronautique, automobile, etc...).

Dans cette étude le problème consiste à optimiser les performances aérodynamiques d'un dispositif hyper-sustentateur par algorithmes génétiques (AG). Il s'agit d'un dispositif aérodynamique permettant d'augmenter la portance d'un avion dans les phases d'atterrissage et de décollage. Le but est de maximiser cette portance calculée par un solveur d'écoulement Eulérien, en couplant ce dernier à un algorithme génétique. Cette classe d'algorithmes est fondée sur les mécanismes de l'évolution formulés par Darwin, et ils peuvent optimiser des fonctionnelles très complexes, en s'affranchissant des problèmes de dérivabilité, tout en étant très robustes. Ils peuvent donc être utiles pour des problèmes liés à la dynamique des fluides, qui sont caractérisés par des calculs fort complexes.

On s'intéresse à deux problèmes d'optimisation spécifiquement : l'optimisation de la position du bec et du volet par rapport au corps principal (à forme constante des trois éléments), et l'optimisation de la forme du bec et du volet.

Un autre objectif consiste à pouvoir spécifier de manière simple l'algorithme génétique lui-même, qui est indépendant du calcul d'écoulement, et à produire aisément un code d'optimisation en C++ couplé à ce dernier. Le but est d'introduire la programmation objet, méthode héritée du génie logiciel, en calcul scientifique.

Au premier chapitre sera présentée la problématique et ses conséquences, puis dans le second chapitre les algorithmes génétiques seront brièvement décrits et enfin les méthodes adoptées et les résultats seront présentés dans le troisième chapitre.

Chapitre 1

Présentation du problème d'optimisation

1.1 Le dispositif hyper-sustentateur

L'atterrissage et le décollage d'un avion sont des phases de vol critiques. En effet à ces moments, la vitesse de l'avion est peu élevée par rapport au régime de croisière et il est donc nécessaire de maximiser la portance afin d'assurer la sustentation de l'avion en vol jusqu'à ce qu'il atterrisse ou atteigne son régime de croisière. Un dispositif dit *hyper-sustentateur* permet d'augmenter la portance de l'appareil lors de l'atterrissage et du décollage. Typiquement, il s'agit d'éléments déployés au moment voulu autour de l'élément principal de l'aile, et repliés contre l'aile pendant le reste du vol. On s'intéresse ici à une configuration comportant trois corps : le bec, l'élément principal et le volet (voir la figure (1.1)). Elle correspond au cas-test D4-25 du projet Esprit EP 25058 DECISION dans lequel l'INRIA a été activement impliqué.

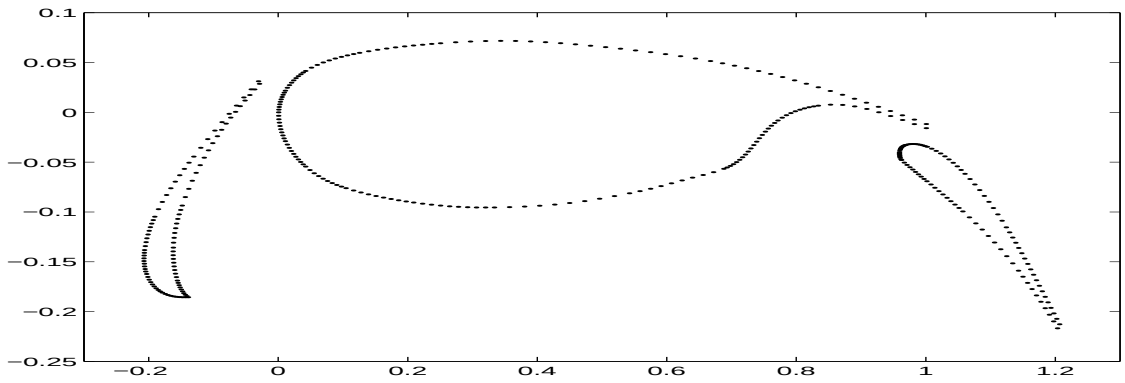

FIG. 1.1 – Allure du profil multiélément

1.2 Le solveur de flot

On cherche à optimiser la portance du dispositif hyper-sustentateur dans le cas de l'atterrissage et du décollage. Il faut donc calculer le flot stationnaire non visqueux compressible autour du profil. Ceci passe par la résolution des équations d'Euler stationnaires en deux dimensions. On se référera à [1] pour tous les détails de la résolution et de son implémentation. Ce calcul constitue une étape vers une optimisation plus réaliste qui incluerait la prise en compte des phénomènes visqueux (équations de Navier-Stokes).

1.2.1 Les équations d'Euler

Les équations d'Euler peuvent s'écrire sous la "forme conservative" suivante :

$$\frac{\partial W}{\partial t} + \frac{\partial F_1(W)}{\partial x} + \frac{\partial F_2(W)}{\partial y} = 0 \tag{1.1}$$

où W est le vecteur des "variables conservatives" qui sont les inconnues du problème :

$$W = \begin{pmatrix} \rho \\ \rho u \\ \rho v \\ E \end{pmatrix} \tag{1.2}$$

où ρ est la masse volumique locale du fluide, u et v les composantes cartésiennes de la vitesse, et E l'énergie totale (interne et cinétique). Les vecteurs $F_1(W)$ et $F_2(W)$ sont les fonctions de flux suivantes :

$$F_1(W) = \begin{pmatrix} \rho u \\ \rho u^2 + p \\ \rho uv \\ u(E+p) \end{pmatrix}, \quad F_2(W) = \begin{pmatrix} \rho v \\ \rho vu \\ \rho v^2 + p \\ v(E+p) \end{pmatrix} \tag{1.3}$$

où p est la pression locale qui s'exprime en fonction des autres variables grâce à l'équation d'état qui, pour un gaz parfait diatomique, ou pour l'air, considéré comme un mélange de tels gaz, est :

$$p = \rho r T = (\gamma - 1)\left(E - \frac{u^2 + v^2}{2}\right) \tag{1.4}$$

où $r = \dfrac{\mathcal{R}}{M}$ est la constante du gaz, M sa masse molaire, $\mathcal{R}$ la constante universelle des gaz parfaits et γ le rapport des capacités calorifiques à pression et volume constants ($\gamma = \dfrac{7}{5}$ pour un mélange de gaz parfaits diatomiques).

Ces équations traduisent, pour un écoulement bidimensionnel permanent de fluide continu compressible, la conservation de la masse (continuité), de la quantité de mouvement (loi de Newton) et de l'énergie (premier principe de la thermodynamique).

Il faut préciser que le modèle utilisé décrit des écoulements de fluides compressibles parfaits c'est à dire *non visqueux*, ce qui constitue bien sûr une approximation. Ce modèle sera critiqué dans la partie (3.4).

1.2.2 Implémentation du solveur

On s'intéresse ici à la méthode numérique utilisée pour résoudre les équations d'Euler, et qui est détaillée dans [1] et résumée dans [2].

Discrétisation du domaine de calcul

On suppose que le domaine de calcul Ω est borné dans $\mathbb{R}^2$. Ce domaine est discrétisé sur un maillage non structuré triangulaire, dont un exemple est donné dans la figure (1.2).

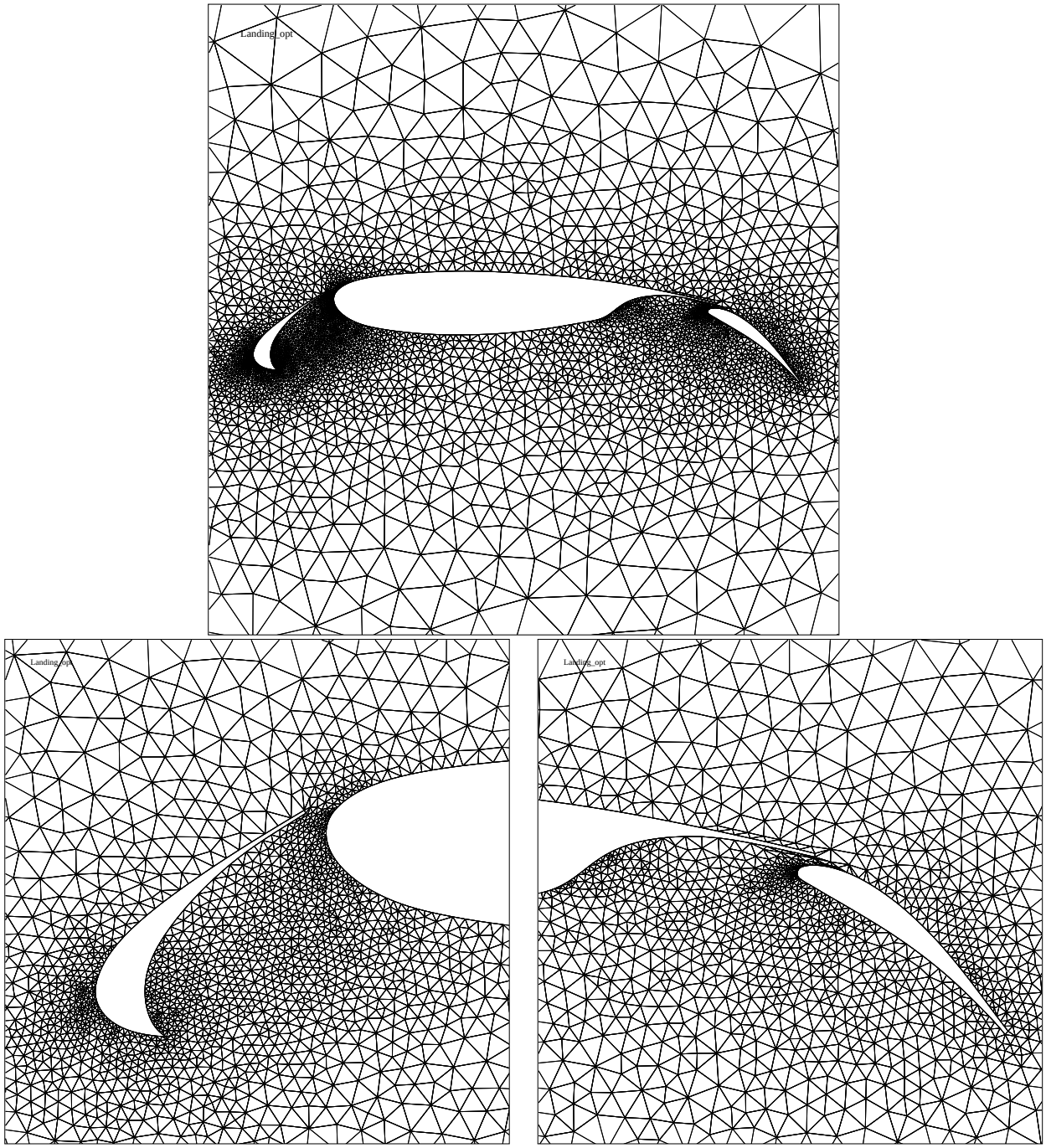

FIG. 1.2 – Exemple de maillage non structuré construit sur une triangulation quelconque

L'utilisation d'un tel maillage présente plusieurs avantages. En effet la construction d'un maillage initial est en soi un problème non trivial, et l'approche "maillage non structuré" simplifie cette construction initiale. De plus cette méthode permet de rendre les nécessaires procédures d'adaptation de maillage à la solution bien plus locales, donc plus faciles à mettre en œuvre, et plus économiques en nombre d'éléments rajoutés.

Les équations d'Euler sont approchées sur un maillage de volumes finis dual construit à partir du maillage initial de la manière suivante : on trace les médianes des triangles, on identifie autour de chaque nœud S_i une cellule de contrôle C_i dont les arêtes sont des portions des médianes des triangles sont le nœud S_i est un sommet (voir la figure (1.3)).

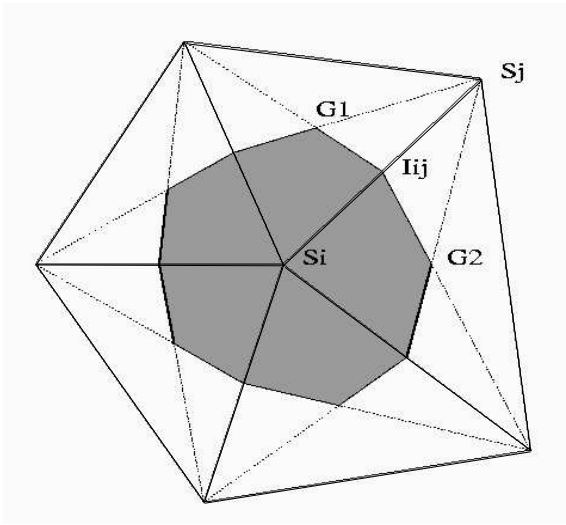

FIG. 1.3 – Cellule du maillage dual de volumes finis construit à partir d'une triangulation quelconque

L'union de toutes ces cellules de contrôle constitue bien une discrétisation duale du domaine de calcul, sur laquelle sont approchées les équations d'Euler par volumes finis décentrés d'ordre deux.

Mise à jour du maillage

Quand une nouvelle configuration du profil multiélément est déterminée, il est évidemment nécessaire de modifier le maillage en conséquence, afin de pouvoir effectuer un nouveau calcul de flot. On utilise ici un modèle pseudo-élastique, dans lequel un ressort virtuel est associé à chaque face du maillage connectant deux nœuds S_i et S_j. Les nœuds situés sur les limites amont et aval du domaine sont fixes, et les nouvelles positions des points du profil sont déterminées par la procédure de paramétrisation du profil. Les nouvelles positions des autres nœuds du maillage sont données en résolvant un problème de déformation pseudo-élastique.

1.3 Optimisation de la position

Le but du dispositif hyper-sustentateur est d'augmenter la portance de l'avion. On cherche donc à optimiser le dispositif de manière à maximiser la portance, et ce pour une vitesse et un angle d'attaque donnés, qui correspondent soit au décollage soit à l'atterrissage. On s'intéresse dans un premier temps à la position du bec et du volet par rapport à l'élément principal. Celle-ci est donnée par trois paramètres qui sont l'angle de déflection (deflection angle) δ, la fente (gap) F et le recouvrement (overlap) R. On a donc en tout six paramètres, tous réels et variant dans des intervalles connus.

Positionnement du bec par rapport à l'élément principal (figure (1.4)) :

- L'angle de déflection (deflection angle) δ_b : il s'agit de l'angle entre la corde de l'élément principal et la droite (Δ). Quand $\delta_b = 0$, le bec est replié sur l'élément principal et (Δ) se confond avec la corde. δ_b est orienté dans le sens trigonométrique direct.
- La fente (gap) F_b : c'est la différence entre l'ordonnée du bord de fuite du bec et l'ordonnée du bord d'attaque de l'élément principal.
- Le recouvrement (overlap) R_b : c'est la différence entre l'abscisse du bord de fuite du bec et l'abscisse du bord d'attaque de l'élément principal.

Positionnement du volet par rapport à l'élément principal (figure (1.5)) :

- L'angle de déflection (deflection angle) δ_v : c'est l'angle entre la corde de l'élément principal et la corde du volet, orienté dans le sens trigonométrique indirect.
- La fente (gap) F_v : c'est la différence entre l'ordonnée du bord de fuite de l'élément principal et l'ordonnée du bord d'attaque du volet.
- Le recouvrement (overlap) R_v : c'est la différence entre l'abscisse du bord de fuite de l'élément principal et l'abscisse du bord d'attaque du volet.

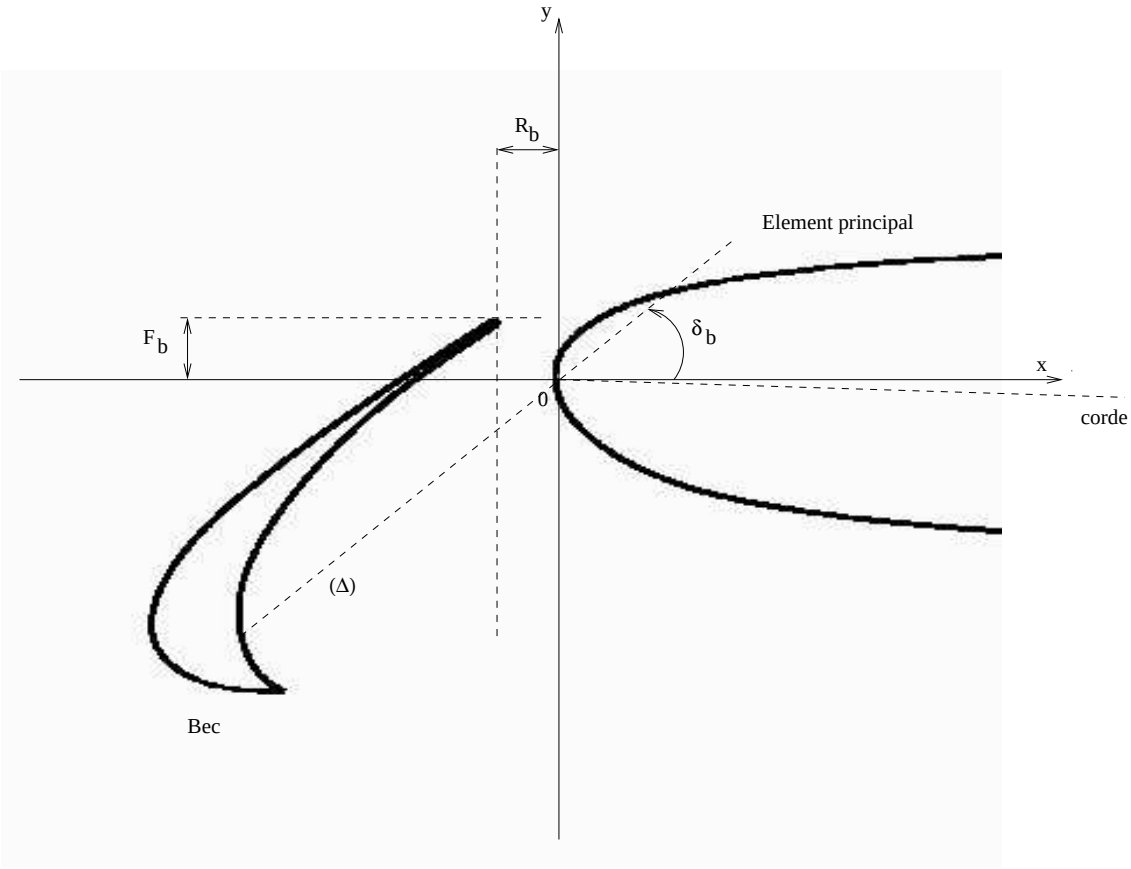

FIG. 1.4 – Positionnement du bec par rapport à l'élément principal

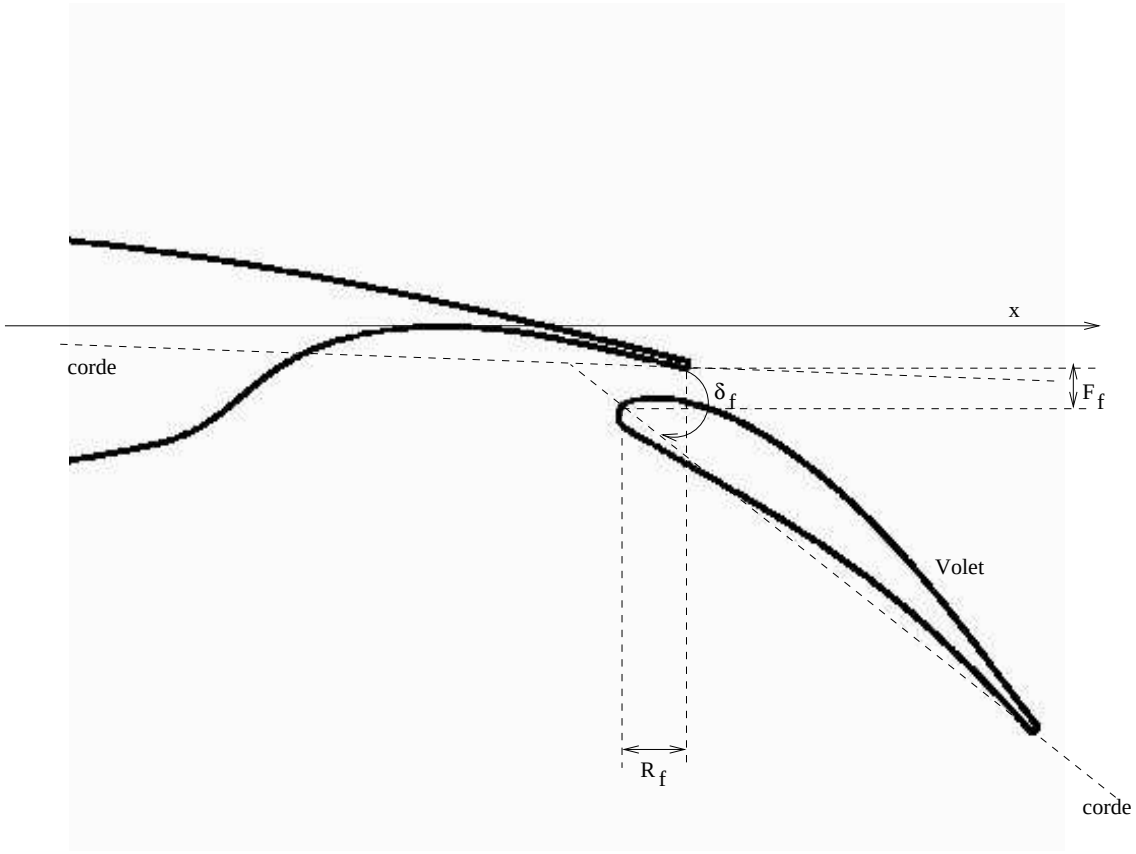

FIG. 1.5 – Positionnement du volet par rapport à l'élément principal

Dans un premier temps, on cherche donc à optimiser un critère réel (la portance du dispositif) selon six variables réelles (δ_b, F_b, R_b, δ_v, F_v, R_v), chacune variant dans un intervalle $[a, b]$ connu.

1.4 Optimisation de la forme

Dans un deuxième temps, on cherchera à améliorer la portance du dispositif selon la position *et* la forme du bec et du volet. La difficulté consiste à paramétriser convenablement cette forme.

1.4.1 Paramétrisation de la forme : difficultés

Rappelons ici que nous travaillons en 2D, sur des profils de multiélément. Le profil initial est donné par le maillage utilisé (cf. partie (1.2.2)), il est donc représenté point par point, chacun des points étant connu par ses coordonnées cartésiennes (x, y). Pour modifier la forme, une méthode naturelle consisterait à faire varier les ordonnées de chaque point dans un certain intervalle $[-\varepsilon; \varepsilon]$. Cette méthode présente deux inconvénients, vis à vis du type d'algorithmes d'optimisation employés, qui sont les algorithmes génétiques (cf. chapitre 2). Premièrement, la trace du maillage sur "la peau" est constituée d'un nombre de points significativement supérieurs au nombre souhaitable de paramètres de conception à optimiser pour que la convergence de l'optimiseur soit raisonnablement bonne (cf. partie (2.1.3)). Deuxièmement, il convient de considérer seulement des formes lisses : pour ces deux raisons la forme est définie par une paramétrisation de Bézier dont les points de contrôle servent de paramètres de conception.

1.4.2 Perturbation de la forme par une courbe de Bézier

Afin de contourner les difficultés citées précédemment, une méthode s'inspirant de la paramétrisation d'un profil (cf. [2]) est employée. Elle repose eur les propriétés des courbes de Bézier. Une courbe de Bézier d'ordre n est une courbe paramétrée définie par :

$$B(t) = \sum_{i=0}^{n} B_{n,i} P_i \quad \text{avec} \quad B_{n,i} = C_n^i t^i (1-t)^{n-i}, \quad C_n^i = \frac{n!}{(n-i)!}$$

On a : $t \in [0, 1]$, $P_i = (x_i, y_i)$ sont les coordonnées des points de contrôle de la courbe. Pour définir une courbe de degré n, $n+1$ points de contrôle sont nécessaires et suffisants. La figure (1.6) donne l'allure d'une courbe de Bézier de degré trois, créée à partir de quatre points de contrôle.

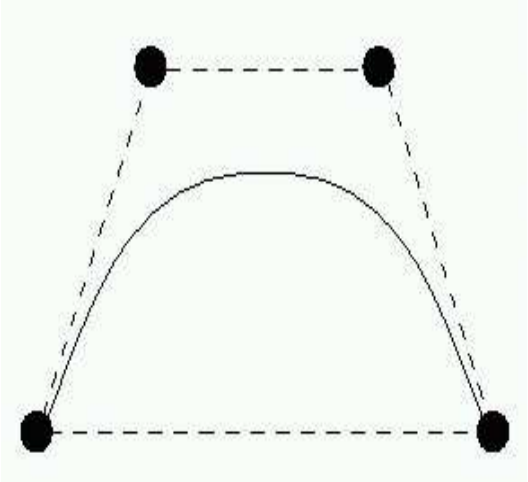

FIG. 1.6 – Allure d'une courbe de Bézier de degré trois

Plutôt que de paramétriser directement la forme des enveloppes du bec et du volet, on définit une *perturbation* par une courbe de Bézier de degré six :

$$\begin{cases} x(t) = \sum_{k=0}^{6} C_6^k t^k (1-t)^{6-k} x_k \\ \delta y(t) = \sum_{k=0}^{6} C_6^k t^k (1-t)^{6-k} \delta y_k \end{cases} \qquad t \in [0,1]$$

On a sept points de contrôle. On choisit de les répartir uniformément sur l'intervalle $[0,1]$. Par conséquent on a $x_k = \frac{k}{6}$. Les points d'abscisse 0 et 1 sont fixes. Supposons que l'on ait n points $(x_i, y_i), i \in \{0, \cdots, n-1\}$ sur l'enveloppe considérée. A ces points on fait correspondre n points répartis uniformément sur l'intervalle $[0,1]$. Ces derniers sont définis par leur abscisse $x'_i = \frac{i}{n-1}$. On résout ensuite pour $i \in \{0, \cdots, n-1\}$:

$$\frac{i}{n-1} = \sum_{k=0}^{6} C_6^k t^k (1-t)^{6-k} x_k$$

Dans le second terme le coefficient de t^k est $\sum_{p=0}^{k}(-1)^{k-p}C_6^pC_{6-p}^{k-p}x_p$. On obtient ainsi une fois pour toutes les t_i tels que $x(t_i) = \dfrac{i}{n-1}$, et on peut calculer les $\delta y_i = \delta y(t_i)$. La nouvelle forme de l'enveloppe est obtenue à partir de l'ancienne par :

$$\begin{cases} x_i^{new} = x_i + \alpha\delta y_i \\ y_i^{new} = y_i + \beta\delta y_i \end{cases} \quad i \in \{0, \cdots, n-1\}$$

α et β sont respectivement les cosinus et sinus de l'angle de déflection moyen du bec ou du volet, selon l'enveloppe considérée. Il y a quatre enveloppes (l'intrados et l'extrados du bec et du volet), il est donc nécessaire de définir quatre courbes de Bézier. Les seuls paramètres nécessaires pour définir ces courbes sont les $\delta y_k, k \in \{1, \cdots, 5\}$, puisque les points de contrôle en 0 et en 1 sont fixes. Cela fait donc 20 paramètres pour la forme du bec et du volet. Chacun d'entre eux varie dans un intervalle $[-\varepsilon; \varepsilon]$ donné.

1.5 Résumé

Si on appelle C_L la portance le problème est :

$$\max_{\delta_b, F_b, R_b, \delta_v, F_v, R_v} C_L$$

avec :

$$C_L = -C_x \sin\alpha + C_y \cos\alpha$$

$$C_x = \frac{\int_\gamma P n_x d\gamma}{\frac{1}{2}\rho V^2 l} \quad \text{et} \quad C_y = \frac{\int_\gamma P n_y d\gamma}{\frac{1}{2}\rho V^2 l}$$

P est la pression, (n_x, n_y) sont les composantes de la normale au profil $\vec{n}$, γ représente le profil, ρ est la densité de l'air, V est le module de la vitesse, l est la longueur du profil et α est l'angle d'incidence du profil.

Le problème consiste donc à optimiser une fonctionnelle complexe, comportant probablement de nombreux optimums locaux. Par ailleurs on envisage dans l'avenir de complexifier encore le problème d'optimisation. On pourrait

par exemple tenir compte aussi de certaines propriétés électromagnétiques, pour satisfaire des contraintes de furtivité (entre autres). Il faut donc utiliser un algorithme d'optimisation aussi robuste possible et aussi général que possible dans sa formulation. Il est en particulier souhaitable qu'il s'appuie sur les valeurs de la fonctionnelle seulement, sans se soucier de la nature des paramètres et de propriétés de dérivabilité et qui puisse éviter de rester dans les optimums locaux, qui peuvent être nombreux. Les Algorithmes Génétiques satisfont de tels critères.

Chapitre 2

Présentation des algorithmes génétiques

2.1 Les algorithmes génétiques simples

2.1.1 Principe

Ces algorithmes sont fondés sur une observation de la nature, et notamment sur les lois de l'évolution formulées par Darwin et qui peuvent se résumer par une formule : la survie du plus adapté. Le principe de fonctionnement des algorithmes génétiques appliqués à des problèmes d'optimisation consiste à simuler les opérateurs auxquels les populations sont soumises dans leur évolution : sélection, croisement, mutation. Parmi une certaine population de solutions (ou points, ou individus) du problème, on ne garde que celles dont l'évaluation est la meilleure selon la fonctionnelle que l'on cherche à optimiser. Une fois cette sélection faite, les solutions restantes peuvent se reproduire, et engendrer ainsi une nouvelle génération de solutions, qui subira ensuite des mutations. L'analogie avec ce qui se passe dans la nature est évidente : dans une population d'individus quelconques de la même espèce, n'arrivent à l'âge où ils peuvent se reproduire que les individus les plus adaptés à l'environnement, les autres sont éliminés. En plus des mécanismes propres à la reproduction, des mutations, bien que plus rares, peuvent amener à l'apparition d'individus originaux. Les algorithmes génétiques reproduisent un schéma analogue : les membres de la

population sont des points du domaine de définition du problème et l'environnement est représenté par la fonctionnelle à optimiser. Les individus les plus adaptés sont ceux pour lesquels la fonctionnelle présente les meilleures valeurs. L'idée est que les individus des générations successives sont de mieux en mieux adaptés à leur environnement. Les algorithmes génétiques produisent ainsi de meilleures solutions à chaque génération vis à vis du ou des critères que l'on veut optimiser.

2.1.2 Fonctionnement des algorithmes génétiques simples

Dans la nature chaque individu est défini par son génome, constitué par tous ses gènes regroupés en chromosomes. Dans les algorithmes génétiques, on aura la même chose : chaque point sera codé sous la forme d'une chaîne de bits, qui représenteront ses gènes. A partir de ces chaînes de bits, appelées aussi chromosomes, on peut retrouver les valeurs des solutions codées. Pour cela, il faut bien entendu choisir un codage approprié. Ensuite des opérateurs simulant les mécanismes de la sélection naturelle sont appliqués à ces chromosomes. Dans les versions simples des algorithmes génétiques, nous avons trois opérateurs : la sélection, le croisement (ou crossover) et la mutation. Ces opérateurs s'appliquent directement sur les chromosomes et dans l'ordre suivant : sélection, croisement et mutation. Une fois ceci fait, on obtient une nouvelle génération de points et on réitère le processus jusqu'à ce que l'on n'ait plus d'amélioration significative ou que l'on ait atteint un nombre maximum d'itérations. Il faut préciser que la population de points traitée par l'algorithme reste constante.

2.1.3 Caractéristiques des AG simples

- Les solutions sont codées sous forme de chaînes de bits, appelées chromosomes. L'algorithme ne travaille que sur ces chaînes. Il faudra choisir le codage de manière à retrouver les valeurs des solutions. Par exemple, supposons que l'on veuille coder avec une précision p un paramètre x réel qui varie sur un intervalle $[a, b]$, avec $b > a$. Soit n le nombre de bits

nécessaires pour coder x, on a :

$$n = \mathrm{E}\left[\frac{\ln\left(\frac{|b-a|}{p}\right)}{\ln 2}\right] + 1$$

Inversement pour passer d'une chaîne de bits de longueur n $(b_0, \cdots, b_{n-1})$ au paramètre x correspondant, on a :

$$x = a + \sum_{i=0}^{n-1} b_i 2^i \frac{|b-a|}{2^n - 1}$$

On peut donc coder plusieurs paramètres sous la forme d'une seule chaîne de bits, du moment que l'on garde pour chacun d'entre eux les informations nécessaires au décodage, soit n, a, et $\frac{|b-a|}{2^n-1}$.

- Les AG traitent simultanément une population de points, et non un point unique, comme le fait la méthode du Gradient par exemple.
- On a seulement besoin de calculer les valeurs de la fonction en chaque point de la population, c'est a dire qu'on n'a pas besoin d'informations additionnelles, sur les derivées par exemple.
- La sélection, le croisement et la mutation sont des opérateurs stochastiques, ce qui signifie que l'optimisation par AG est *non-déterministe.*
- La complexité temporelle (c'est à dire le nombre d'opérations élémentaires, ou flops) des AG séquentiels est en $O(N \times l)$ ou $O(NlogN + N \times l)$, selon la méthode de sélection adoptée, où N est la taille de la population et l la longueur des chromosomes (cf. [3]). Ainsi, cette complexité dépend directement du nombre de paramètres, puisque ces derniers conditionnent la longueur des chromosomes. Il faut noter que ces opérations élémentaires ne concernent pas l'évaluation proprement dite.

2.1.4 Les opérateurs génétiques simples

Ils sont au nombre de trois : la sélection, le croisement et la mutation. Pour passer d'une génération à l'autre, on applique ces opérateurs à l'ensemble des points et dans l'ordre suivant : sélection, croisement et mutation.

La sélection

Elle se fonde sur la valeur d'adaptation des solutions. Cette valeur d'adaptation dépend de la fonctionnelle que l'on veut optimiser et du type d'optimisation que l'on veut faire : maximisation ou minimisation. On trouve plusieurs versions de cet opérateur, les plus simples sont la roulette et le tournoi.

La roulette : Selon leur valeur d'adadaptation, chaque point de la population reçoit une probabilité de sélection. Meilleure est cette valeur, plus grande est cette probabilité. On tire ensuite au hasard n points, où n est la taille de la population. On obtient n nouveaux points, où les meilleures solutions sont plus représentées.

Par exemple supposons que l'on ait la situation suivante : une population de quatre individus dotés d'une valeur d'adaptation positive que l'on cherche à maximiser. On obtient facilement la probabilité de sélection pour chaque individu :

Individu	**Adaptation**	**Probabilité de sélection**
1	169	0,144
2	576	0,492
3	64	0,055
4	361	0,309
	total = 1170	total = 1

TAB. 2.1 – Exemple de sélection par roulette

Le tournoi : On fait un tirage aléatoire avec remise de deux points de la population, on compare leur évaluation, et le meilleur des deux est sélectionné. On recommence jusqu'à obtenir une nouvelle population de la taille voulue. Il s'agit d'une stratégie élitiste (cf. partie (2.4.3)).

Remarque : Avec ces deux méthodes, on remarque qu'un individu peut être reproduit en plusieurs exemplaires d'une génération à l'autre. Nous verrons après que cela peut être un inconvénient (cf. partie (2.3.2)).

Le croisement à un point

Chaque point de la population peut être sélectionné pour subir un croisement. La probabilité pour qu'un point subisse un croisement est généralement élevée. Les points sélectionnés sont appariés de manière aléatoire. Un entier k est choisi au hasard entre 1 et l-1, où l est la longueur des chromosomes. On échange ensuite tous les bits des deux chaînes entre les positions k+1 et l inclus. On obtient deux nouveaux chromosomes qui vont remplacer leurs parents dans la nouvelle population (voir la figure (2.1)). Cet opérateur permet de créer de nouvelles solutions à partir de celles qui ont été sélectionnées précédemment.

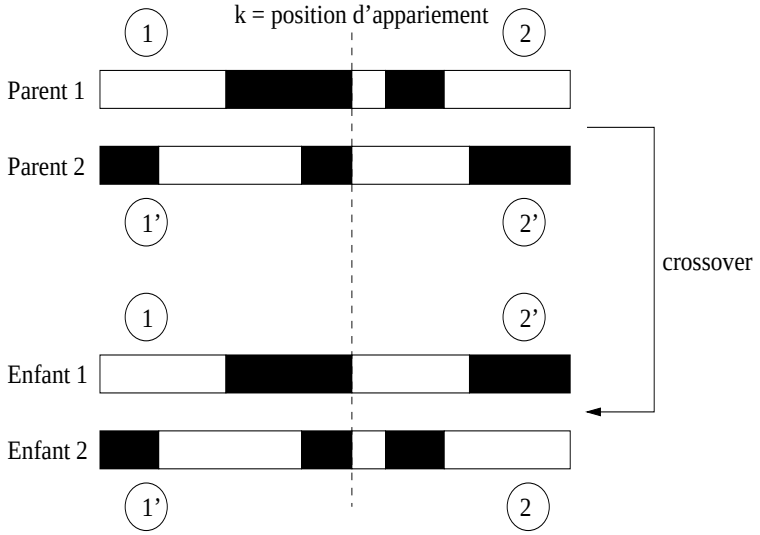

FIG. 2.1 – Exemple de croisement à un point

Avec cet opérateur, on espère, par la recombinaison de deux bons individus , en obtenir deux autres encore meilleurs. Ceci repose sur une observation de mécanismes naturels, mais aussi et surtout sur la théorie des schèmes (cf. partie (2.2)).

La mutation

Cet opérateur est appliqué à chaque bit des chromosomes avec une probabilité faible. Il change un 1 en 0 et vice-versa. Il permet d'introduire un facteur

aléatoire dans les solutions générées, et d'élargir ainsi l'espace des solutions explorées.

2.1.5 Paramètres d'un AG simple

Les paramètres pour un AG simple sont les suivants :
- La taille de la population.
- Le nombre maximum de générations.
- La probabilité de croisement, généralement élevée.
- La probabilité de mutation, généralement faible.

Le problème de la paramétrisation de l'algorithme se pose ici. Ces paramètres influencent notablement les performances de l'AG. Dans sa thèse (cf. [4]) De Jong suggère de choisir une probabilité de croisement élevée, une probabilité de mutation faible (à peu près égale à l'inverse de la taille de la population), et une population de taille modérée. Ainsi des paramètres typiques pour un AG simple sont :
- Pmut = 0,05 (probabilité de mutation)
- Pcross = 0,9 (probabilité de croisement)
- PSize = 30 (taille de la population)

Il faut cependant souligner que la paramétrisation d'un algorithme génétique relève beaucoup de l'intuition, de plus cette paramétrisation dépendra souvent de la nature de l'optimisation et du problème étudié.

2.2 Les schèmes

2.2.1 Définitions

Un schème est une chaîne formée avec un jeu de trois caractères : $\{0\ ,\ 1\ ,\ *\}$. $*$ représente indifféremment un 0 ou un 1. Un schème représente donc un ensemble de chaînes de bits. Un schème correspond à 2^r chaînes, où r est le nombre de $*$. Une chaîne de longueur m peut être représentée par 2^m schèmes. dans une population de n chaînes de longueur m il y a donc entre 2^m et $n2^m$ schèmes représentés.

Définition 1 ([5]) *L'ordre d'un schème est le nombre de positions instanciées (différentes de $*$) du schème.*

Définition 2 ([5]) *La longueur utile d'un schème est la distance entre la première et la dernière position instanciée du schème.*

2.2.2 Le théorème des schèmes (Théorème Fondamental des Algorithmes Génétiques)

Définition 3 *La valeur d'adaptation d'un schème est la moyenne des valeurs d'adaptation des chaînes correspondantes au schème. On dit d'un schème qu'il est performant si sa valeur d'adaptation est supérieure à la moyenne des valeurs d'adaptation de toutes les chaînes de la population.*

Si il y a p chaînes $(v_1, ..., v_p)$ *correspondantes au schème S alors la valeur d'adaptation de S est :*

$$eval(S) = \sum_{i=1}^{p} \frac{eval(v_i)}{p}$$

Notons $m_t(H)$ le nombre d'exemplaires du schème H à la date t dans une population contenant n individus. Après reproduction, on a :

$$m_{t+1}(H) = m_t(H)\frac{f_t(H)}{f_t} \tag{2.1}$$

où $f_t(H)$ représente la valeur d'adaptation moyenne du schème H à la date t et f_t est la valeur d'adaptation moyenne de toutes les chaînes de la population à la date t (cf. [5]).

Supposons ensuite que l'on effectue après la sélection un croisement à un point, tel que défini précedemment (cf. partie (2.1.4)). Soit la chaîne A de longueur 7 et deux schèmes H_1 et H_2 qui correspondent tous deux à la chaîne A :

$$\begin{array}{lccc|cccc} A= & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ H_1= & * & 1 & * & * & * & * & 0 \\ H_2= & * & * & * & 1 & 0 & * & * \end{array}$$

Le | correspond au point de césure des chromosomes pour le croisement à un point. A moins que la chaîne appariée avec A ne lui soit identique aux positions instanciées du schème (ce qui est peu probable), le schème H_1 sera détruit car le 1 en position 2 et le 0 en position 7 sont placés dans deux

descendants distincts. Avec le même point de césure, le schème H_2 survivra car le 1 en position 4 et le 0 en position 5 seront tous deux préservés dans le même descendant. En bref, le schème H_1 a moins de chance de survivre au croisement que le schème H_2 parce que le point de césure a plus de chances de tomber entre les positions instanciées extrêmes.

Pour quantifier ces observations, on remarque que le schème H_1 a une longueur utile de 5. Si le point de croisement est choisi uniformément au hasard parmi les $l-1=7-1=6$ points possibles ($l=7$ est la longueur du schème), alors le schème H_1 a une probabilité de destruction $p_d = \dfrac{\delta(H_1)}{l-1} = \dfrac{5}{6}$ où $\delta(H_1)$ représente la longueur utile du schème H_1. Il a donc une probabilité de survie $p_s = 1 - p_d = \dfrac{1}{6}$. De même, le schème H_2 a une longueur utile $\delta(H_2) = 1$, et il est détruit uniquement quand le point de césure est entre les positions 4 et 5, et donc sa probabilité de survie est de $\dfrac{5}{6}$. Plus généralement, une borne inférieure de la probabilité p_s de survie au croisement peut être calculée pour un schème quelconque. Comme un schème survit quand le point de croisement tombe à l'extérieur de sa longueur utile, la probabilité de survie lors d'un croisement simple est $p_s = 1 - \dfrac{\delta(H)}{l-1}$. Ceci n'est qu'une borne inférieure, car il ne faut pas oublier que l'appariement peut reconstituer le schème après la césure. Si le croisement a lieu avec une probabilité p_c lors d'un appariement quelconque, la probabilité de survie peut être exprimée ainsi (cf. [5]) :

$$p_s \geq 1 - p_c \frac{\delta(H)}{l-1}$$

L'effet combiné de la reproduction et du croisement sur le nombre de schèmes attendus à la génération suivante, en supposant les deux opérations indépendantes, donne donc l'estimation suivante (cf. [5]) :

$$m_{t+1}(H) \geq m_t(H) \frac{f(H)}{f} \left(1 - p_c \frac{\delta(H)}{l-1}\right) \tag{2.2}$$

On voit donc que le schème H se développe ou dépérit selon un facteur multiplicatif. En considérant le croisement et la reproduction, ce facteur dépend de deux choses : le fait que le schème soit au-dessus ou en-dessous de la

moyenne de la population et le fait que le schème ait une longueur utile relativement courte ou longue. Les schèmes ayant à la fois une adaptation élevée et une courte longueur utile vont être échantillonnés à une fréquence croissant exponentiellement.

Il ne reste plus qu'à prendre en compte la mutation. En utilisant la définition précédente, il est évident qu'un schème survivra si aucune de ses positions instanciées n'est modifiée. Il y a $o(H)$ positions instanciées ($o(H)$) est l'ordre du schème) et la probabilité de survie de chacune est $(1 - p_m)$, où p_m est la probabilité de mutation d'un gène. La probabilité de survie à la mutation du schème est donc $(1 - p_m)^{o(H)}$. Quand p_m est petit ($p_m \ll 1$) cette expression peut être approximée par $1 - o(H)p_m$. En considérant les 3 opérateurs, reproduction, croisement et mutation, l'estimation du nombre de représentations du schème H à la génération suivante est (cf. [5]) :

$$m_{t+1}(H) \geq m_t(H)\frac{f(H)}{f}\left(1 - p_c\frac{\delta(H)}{l-1} - o(H)p_m\right) \tag{2.3}$$

On peut donc déduire de tout ceci le théorème suivant :

Théorème 1 (Théorème des schèmes, [5]) *Les schèmes performants, de petite longueur utile et d'ordre faible correspondent à un nombre exponentiellement croissant de chaînes au fur et à mesure des générations d'un AG.*

Et ce théorème amène le postulat ci-dessous :

Hypothèse 1 (Hypothèse des briques élémentaires, [5]) *Un AG recherche des performances quasi-optimales par la juxtaposition de schèmes de petite longueur utile, d'ordre faible et performants, appelés briques élémentaires.*

En clair, un AG construit des solutions performantes à partir de morceaux de chaînes qui sont potentiellement performantes.

2.3 Avantages et inconvénients des AG

2.3.1 Les avantages des AG

– Les AG opèrent au niveau du codage des paramètres sans se soucier de leur nature, donc ils s'appliquent à de nombreuses classes de problèmes,

problèmes qui dépendent éventuellement de plusieurs paramètres de natures différentes (booléens, entiers, réels, fonctions...).
- Pour les mêmes raisons, un AG est dans l'idéal totalement indépendant de la nature du problème et de la fonctionnelle à optimiser, car il ne se sert que des valeurs d'adaptation, qui peuvent être très différentes des valeurs de la fonction optimisée, même si elles sont calculées à partir de cette dernière.
- Potentiellement, les AG explorent tout l'espace des points en même temps, ce qui limite les risques de "tomber" dans des optimums locaux.
- Les AG ne se servent que des valeurs de la fonctionnelle pour optimiser cette dernière, il n'y a pas besoin d'effectuer de coûteux et parfois très complexes calculs de dérivées par exemple.
- Les AG présentent une grande robustesse, c'est a dire une grande capacité à trouver les optimums globaux des problèmes d'optimisation, même non convexes et multi-critères.
- Enfin les AG sont aisément parallélisables (cf. partie (2.4.3)).

2.3.2 Les inconvénients des AG

- Les AG ne sont encore actuellement pas très efficaces en coût (ou vitesse de convergence), vis à vis de méthodes d'optimisation plus classiques.
- Parfois les AG convergent très vite vers un individu particulier de la population dont la valeur d'adaptation est très élevée (on parle de *super-individu*).
- Le respect des contraintes de domaine par les solutions codées sous forme de chaînes de bits pose parfois problème. Il faut bien choisir le codage, voire modifier les opérateurs.
- En pratique l'efficacité d'un AG dépend souvent de la nature du problème d'optimisation. Selon les cas, le choix des paramètres et des opérateurs sera souvent critique, mais aucune théorie générale ne permet de connaître avec certitude la bonne paramétrisation. Il faudra faire plusieurs expériences pour s'en approcher.

2.4 Améliorations de la technique de base

Un ensemble de modifications de l'algorithme de base ont été codées et testées. Ces modifications qui concernent principalement l'algorithme d'optimisation mais aussi son codage informatique (parallélisation) sont détaillées dans les paragraphes de cette section.

2.4.1 La stratégie des niches

Motivation

Dans certaines conditions les AG sont susceptibles de converger prématurément vers un petit nombre d'individus qui ont une valeur d'adaptation très élevée, réduisant ainsi la diversité de la population. Ceci se produit notamment lorsqu'on trouve de tels individus dans une population initiale de congénères médiocres, ce qui est relativement courant. Du fait de la nature des opérateurs de sélection, la population deviendra rapidement constituée d'un nombre restreint d'individus différents, reproduits en nombreux exemplaires. Ceci est un inconvénient, car la robustesse des AG repose sur la diversité de la population des solutions. Si cette diversité décroît, cela augmente d'autant les risques pour l'algorithme de se retouver piégé dans un optimum local. Dans le pire des cas, la population finale sera constituée de copies du même individu.

Préservation de la diversité par nichage

Pour préserver la diversité des individus dans une population, il faudrait que des sous-populations stables se forment près de chaque optimum de la fonction. Ceci peut être fait en adoptant une stratégie de "niches écologiques", décrite dans [6], et qui simule certains mécanismes naturels. Un environnement naturel est en effet très souvent divisé en sous-environnements exploités par des sous-populations distinctes. On a une séparation de l'environnement et des organismes exploitant cet environnement en différents sous-ensembles.

Pour simuler ce mécanisme, il est possible de procéder au partage de la valeur d'adaptation entre les individus similaires. Supposons que l'on peut définir une distance entre deux individus de la population (on peut par exemple utiliser la distance de Hamming, qui est simplement le nombre de bits différents

entre deux chromosomes). On définit une fonction de partage de la manière suivante :

$$Sh(d(j)) = \begin{cases} 1 - \left(\frac{d(j)}{\sigma}\right)^{\alpha} & \text{si } d(j) < \sigma \\ 0 & \text{si } d(j) \geq \sigma \end{cases}$$

Si $\sigma = 10$ et $\alpha = 1$, on a une fonction de partage triangulaire dont l'allure est donnée dans la figure (2.2).

σ est le rayon de la niche. Ainsi, pour un individu i donné, son adaptation avant partage f_i est divisée par la somme des individus de toute la population, pondérés par la fonction de nichage définie précédemment. La valeur d'adaptation qui sera utilisée pour la sélection suivante sera donc :

$$f_i^p = \frac{f_i}{\sum_{j=1}^{PSize} Sh(d(i,j))}$$

Ainsi, quand beaucoup d'individus sont dans le même voisinage, ils contribuent mutuellement aux comptes de partage des uns et des autres, ce qui dégrade les valeurs d'adaptation. En conséquence, le mécanisme limite le développement incontrôlé de certains individus. Il convient de préciser que le choix de σ influe beaucoup sur l'efficacité de la technique. Ce choix dépend essentiellement de la nature du problème.

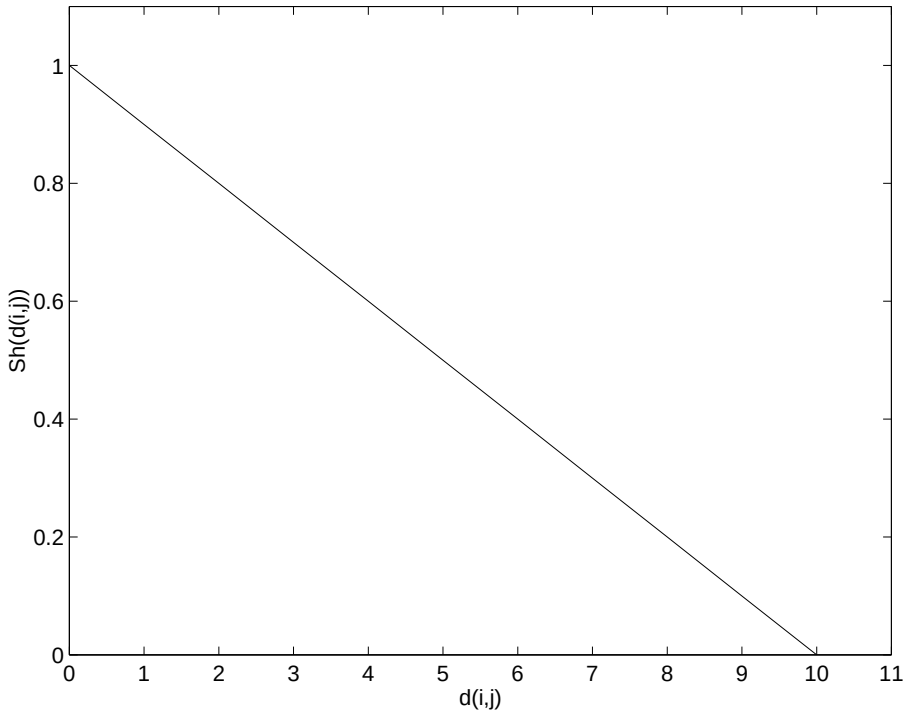

FIG. 2.2 – Exemple de fonction de partage triangulaire

2.4.2 Transformation linéaire de la fonction d'adaptation

Motivation

Il s'agit d'une autre technique visant à éviter une convergence prématurée de l'algorithme génétique. On a vu qu'elle pouvait se produire lorsque l'on trouve quelques individus très adaptés dans une population plutôt médiocre. Ces individus s'attribueraient une proportion importante de la population finie d'une génération, d'où convergence prématurée. Un autre problème est susceptible d'apparaître plus tard dans l'exécution. En effet, il est possible qu'il y ait encore une bonne diversité dans la population, cependant l'adaptation moyenne risque d'être proche de l'adaptation du meilleur individu. Si l'on ne fait rien, les individus moyens et les meilleurs reçoivent à peu près le même nombre de copies dans les générations suivantes, et la survie du plus adapté,

qui est nécessaire à l'optimisation, devient une exploration aléatoire parmi les individus médiocres.

La technique de la transformation linéaire

Cette technique décrite dans [7] permet de résoudre le problème soulevé précédemment. Il consiste à transformer la fonction d'adaptation f en une autre fonction d'adaptation f' par le biais d'une transformation linéaire : $f' = af + b$.

Les coefficients a et b peuvent être choisis comme on veut, mais la moyenne de f' doit être égale à la moyenne de f. Ainsi chaque individu d'adaptation moyenne aura un nombre de copies attendu égal à un. Pour contrôler le nombre de descendants de l'individu ayant l'adaptation brute maximale, on choisit la transformation de façon à avoir $f'_{max} = Cf_{moy}$, où C est l'espérance du nombre de copies souhaité pour le meilleur individu. Pour obtenir ce résultat, on peut choisir a et b de la façon suivante :

$$a = \frac{(C-1)f_{moy}}{f_{max} - f_{moy}} \quad b = \frac{f_{moy}(f_{max} - Cf_{moy})}{f_{max} - f_{moy}}$$

Cependant il est possible avec cette transformation, que des adaptations très faibles deviennent négatives, notamment quand des valeurs brutes sont très en-dessous de la moyenne et du maximum, eux-mêmes relativement proches. Cela n'est pas tolérable. Avec les coefficents a et b définis précédemment, cela se produit si $f_{min} < \frac{Cf_{moy} - f_{max}}{C-1}$. Dans ce cas, on s'arrange pour conserver la moyenne de l'adaptation brute et pour avoir $f'_{min} = 0$. On redéfinit alors a et b ainsi :

$$a = \frac{f_{moy}}{f_{moy} - f_{min}} \quad b = \frac{-f_{min}f_{moy}}{f_{moy} - f_{min}}$$

De cette façon, une simple transformation aide à empêcher une domination précoce des individus extraordinaires, et à encourager plus tard une compétition saine entre ceux qui sont presque égaux.

2.4.3 Autres améliorations réalisées

Le changement d'échelle en puissance

Avec cette méthode, l'adaptation transformée prend simplement la valeur d'une certaine puissance de l'adaptation brute : $f' = f^k$. Le choix de k dépend beaucoup du problème (voir [8]).

Elitisme et super-élitisme

L'élitisme, dans sa version simple, consiste à garder en mémoire le meilleur individu jamais rencontré au cours de l'algorithme. Par exemple, dans une population d'individus, le dernier sera toujours le meilleur jamais rencontré. Après la sélection, le croisement et la mutation, l'élitisme consiste à remplacer le pire individu de la nouvelle population par le meilleur de l'ancienne, si celui-ci est effectivement meilleur.

Le super-élitisme utilise le même principe, mais au lieu de garder en mémoire un seul individu, il en garde plusieurs. le nombre peut être déterminé de plusieurs façons, par exemple en spécifiant une proportion de la population totale. Par exemple, avec un super-élitisme de 10%, une fois la nouvelle population générée, on remplace les 10% de moins bons points de la nouvelle population par les 10% meilleurs de l'ancienne, quand ces derniers sont effectivement meilleurs.

Avec l'élitisme ou le super-élitisme, on s'assure que l'adaptation du meilleur individu ne peut que croître et l'algorithme est ainsi plus "stable".

L'échantillonnage stochastique sans remplacement

Il s'agit d'une méthode de sélection plus évoluée que celles qui ont été vues précédemment. Elle est donnée dans [8]. Pour chaque individu i, on calcule son nombre de copies attendues dans la population suivante. Ce nombre est $n_i = PSize \times \frac{f_i}{\bar{f}}$, où $PSize$ est la taille de la population, f_i est l'adaptation de l'individu i, et $\bar{f}$ est la moyenne des adaptations de tous les individus de la population. Chaque individu se voit accordé un nombre de copies égal à la partie entière de n_i. Les places restantes sont attribuées selon la valeur des parties décimales des n_i. Ces parties décimales sont utilisées comme des

probabilité de sélection. Par exemple, un individu avec un nombre de copies attendu de 1,4 aura dans la population suivante une copie avec certitude et une autre copie avec une probabilité de 0,4.

Parallélisation

La parallélisation d'un code d'optimisation par algorithme génétique est dans son principe assez simple. En effet un AG est, par sa structure, intrinsèquement parallélisable. On a une population initiale de n solutions, ces dernières sont évaluées, et à partir de la population et des valeurs d'adaptation des individus, une nouvelle population est générée. On répète ensuite le processus jusqu'à satisfaire un critère quelconque. On voit que d'une génération à l'autre, l'algorithme procède à l'évaluation des individus de la population courante. Cette évaluation est directement liée à la fonctionnelle à optimiser (en général, c'est la valeur $f(x_i)$, où f est la fonctionnelle et x_i la solution correspondant à l'individu i après décodage). Ces évaluations servent à calculer les valeurs d'adaptation (quand elles ne sont pas égales). Or les évaluations des individus sont tout à fait indépendantes, et rien n'interdit qu'elles soient faites par des processus distincts. L'AG peut ainsi gagner beaucoup en efficacité, surtout si l'évaluation des solutions est longue.

2.4.4 La stratégie de Nash

Le principe

La stratégie de Nash est issue de la théorie des jeux et de l'Economie. Elle repose sur le principe de la compétition entre joueurs. Supposons que deux joueurs 1 et 2 obtiennent un certain score selon leurs actions dans le jeu, et que chacun essaye d'obtenir un meilleur score que l'autre. On suppose aussi que chaque joueur connaît les actions de l'autre. 1 et 2 vont chacun tenter de maximiser leur score, en tenant compte des actions de l'adversaire. Au fur et à mesure des tours de jeu, les scores des deux joueurs deviennent de plus en plus élevés, jusqu'à ce qu'ils deviennent stables sans évoluer davantage. C'est ce qu'on appelle l'équilibre de Nash.

Application aux AG

Il est aisé d'appliquer la stratégie de Nash avec des algorithmes de type AG. Supposons que l'on ait le problème d'optimisation suivant : $\max_{a,b} F(a,b)$. Si on utilise un AG simple, celui-ci se servira d'une population de doublets (a,b) qu'il fera évoluer afin d'obtenir le meilleur individu possible, c'est à dire celui pour lequel $F(a,b)$ est le plus élevé. Appliquer la stratégie de Nash à ce problème revient à faire tourner deux AG devant chacun optimiser une quantité différente. On parle alors de joueurs et on a :

$$\begin{aligned} 1^{er}\text{joueur} &: \max_{a} F(a,b^*) \\ 2^{eme}\text{joueur} &: \max_{b} F(a^*,b) \end{aligned}$$

où a^* est le paramètre optimal trouvé au tour précédent par le premier joueur et b^* celui trouvé par le second joueur. En fait chaque AG optimise la fonctionnelle selon un paramètre, et ils s'échangent toutes les n générations le meilleur paramètre qu'ils ont trouvé jusqu'ici (n est alors appelé la fréquence de Nash). Au bout d'un moment, aucun AG n'est plus capable d'améliorer encore sa fonctionnelle. On dit alors qu'on a atteint l'équilibre de Nash. On a alors :

$$F(a^*,b^*) = \max_{a} F(a,b^*) = \max_{b} F(a^*,b)$$

Il est également possible avec cette stratégie de trouver un compromis entre deux problèmes d'optimisation différents utilisant les mêmes paramètres. Soit les deux problèmes d'optimisation :

$$\max_{a,b} F_1(a,b) \quad \text{et} \quad \max_{a,b} F_2(a,b)$$

On peut utiliser une stratégie de Nash avec deux joueurs tel que :

$$\begin{aligned} 1^{er}\text{joueur} &: \max_{a} F_1(a,b^*) \\ 2^{eme}\text{joueur} &: \max_{b} F_2(a^*,b) \end{aligned}$$

Si les deux problèmes d'optimisation sont "concurrents", c'est à dire que les solutions optimales doivent être différentes, alors l'équilibre de Nash (a^*,b^*) est un compromis entre les optimums des deux problèmes séparés.

Intérêts de la stratégie de Nash

De nombreux atouts de cette stratégie repose sur le fait que chaque joueur travaille avec des chromosomes plus courts que l'AG simple, car ils ont chacun moins de paramètres. Dans l'exemple précédent les deux joueurs travaillent avec deux fois moins de paramètres que l'AG simple. En pratique chaque joueur converge plus vite vers l'optimum que l'AG simple, et ce pour deux raisons principales :

- Le domaine à explorer étant plus restreint, il est plus rapide de trouver les optimums
- Chaque joueur a une partie du chromosome optimisée et fixe donc les briques élémentaires performantes sont moins facilement détruites (cf. Hypothèse 1).

La convergence vers l'équilibre de Nash est donc plus rapide en nombre de générations, et cet équilibre produit donc une valeur supérieure de $F_(a, b)$ (à nombre de générations donné).

Si on veut une solution qui soit "bonne" sans être optimale pour deux problèmes concurrents, la stratégie de Nash est intéressante.

De plus cette stratégie est très intéressante algorithmiquement car intrinsèquement parallélisable. En effet mis à part les informations qu'ils s'échangent, les deux joueurs sont totalement indépendants et donc susceptibles de constituer deux processus distincts. Si on fait les évaluations de solutions parallèlement (cf. partie (2.4.3)), on obtient alors deux niveaux de parallélisme d'où un programme potentiellement très efficace.

Le schéma de principe d'un algorithme génétique est donné dans la figure (2.3). Nous allons voir dans le prochain chapitre comment appliquer les algorithmes génétiques au problème d'optimisation posé dans le chapitre 1.

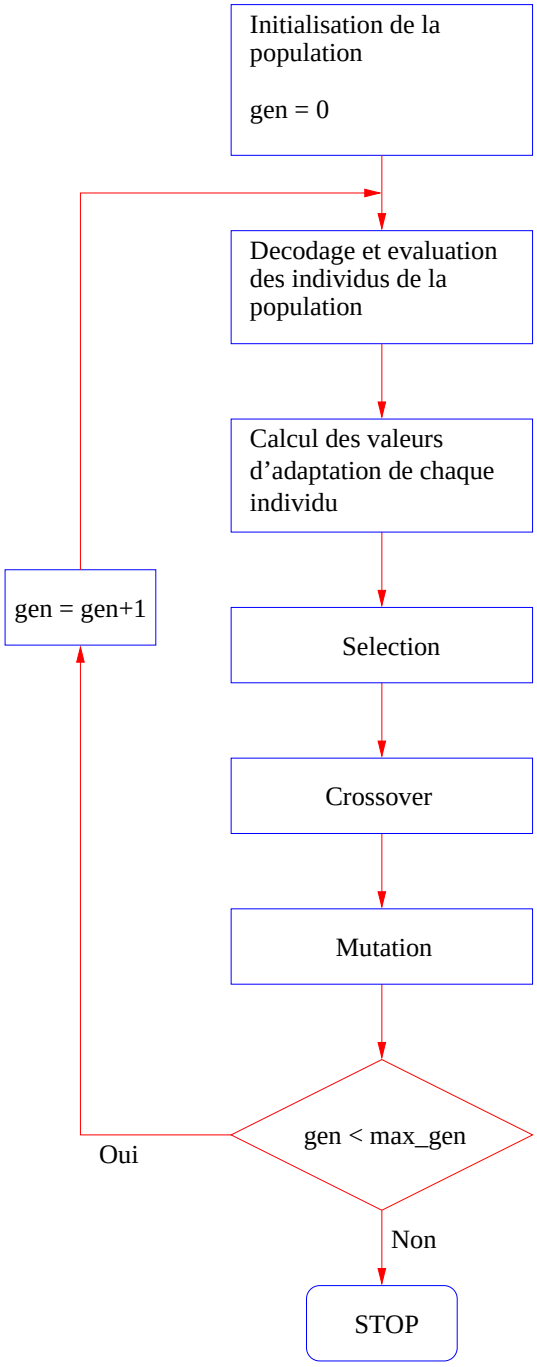

FIG. 2.3 – Organigramme d'un AG simple

Chapitre 3

Expériences numériques et résultats

3.1 Expérience préliminaire

Une première étape a consisté à se familiariser avec le fonctionnement des algorithmes génétiques en réutilisant un code existant, écrit en Fortran 77. Ce code optimise la position du bec et du volet d'un dispositif hyper-sustentateur (décrit dans la partie (1.1)) afin de maximiser sa portance.

3.1.1 Description succincte du code

Le code est composé de deux grandes procédures distinctes.

La première est le solveur Eulérien proprement dit, dont le principe de fonctionnement est résumé dans la partie (1.2). Cette procédure recoit une série de n jeux de six paramètres, où n est la taille de la population pour l'AG. Chaque jeu de six paramètres définit une configuration du profil et représente donc un individu de la population de l'AG. Chaque individu est donc représenté par les paramètres $(\delta_b, F_b, R_b, \delta_v, F_v, R_v)$ (cf. partie (1.3)). Un calcul d'écoulement est fait sur chaque profil, et la procédure renvoie donc pour chaque individu une valeur d'adaptation égale à $-C_L$, où C_L désigne la portance.

La seconde procédure applique à la population d'individus les opérateurs propres à l'algorithme génétique. Elle modifie tout d'abord l'adaptation des

individus, conformément à la stratégie des niches (cf. partie (2.4.1)), si l'utilisateur a choisi d'utiliser cette méthode. Puis elle applique la sélection, qui peut se faire par roulette ou par tournoi (cf. partie (2.1.4)). Enfin le croisement est appliqué et une nouvelle population de n individus est générée. Il faut préciser que la stratégie élitiste est employée, et que l'adaptation est en fait minimisée, car égale à $-C_L$.

Deux ajouts ont été apportés à ce code de départ : une procédure réalisant la transformation linéaire de la fonction d'adaptation (cf. partie (2.4.2)), et une procédure de sélection par échantillonnage stochastique sans remplacement (cf. partie (2.4.3)).

Le fonctionnement général de ce code est donné dans l'annexe A. Il faut cependant préciser ici que deux calculs d'écoulement sont effectués avec une précision plus élevée avant et après la boucle d'optimisation proprement dite : le premier sur le profil initial et le second sur le profil optimal. Dans la boucle principale, des calculs moins convergés en temps sont faits, ce qui suffit à l'algorithme génétique.

3.1.2 Paramètres du solveur et définition du problème

Les paramètres principaux pour le solveur sont le Mach et l'angle d'incidence. Le Mach est le rapport de la vitesse de l'appareil avec la vitesse du son. L'angle d'incidence est l'angle que fait la corde du corps principal avec la direction de l'écoulement. Selon le cas de vol étudié, ces paramètres diffèrent. On aura donc :
Cas de l'atterrissage : Mach = 0,12 ; incidence = 17,18 degrés
Cas du décollage : Mach = 0,25 ; incidence = 10 degrés
Le maillage est constitué de 4449 nœuds pour 8491 triangles.

Le problème d'optimisation est le suivant :

$$\min_{\delta_b, F_b, R_b, \delta_v, F_v, R_v} -C_L$$

3.2 Reproduction de l'expérience avec EASEA

3.2.1 Présentation d'EASEA

On l'a vu avec le code Fortran 77, le codage des fonctions et opérateurs propres à l'AG n'est pas évident. Or ces derniers sont tous groupés dans une procédure totalement distincte du solveur lui-même, qui constitue la fonction d'évaluation. Une idée serait donc d'avoir un programme ou langage relativement simple dédié à la spécification de l'algorithme génétique, ce qui éviterait d'avoir à tout faire. Des bibliothèques sont donc apparues, qui contiennent des outils, fonctions et opérateurs facilitant l'écriture d'algorithmes de type AG. Malheureusement, la plupart de ces bibliothèques, programmées en C++, demandent une bonne connaissance de ce langage et de la programmation objet en général.

Afin de "démocratiser" les AG, l'Action de Recherche Coopérative EVOLAB (EVOlutionary LABoratory) a pour objectif de proposer une interface graphique qui permettrait à des non-spécialistes de spécifier facilement leurs propres algorithmes génétiques. EASEA (EAsy Specification for Evolutionary Algorithms) est un programme développé par Pierre Collet dans le cadre de l'action EVOLAB, et qui constitue une étape dans la réalisation de cette interface. EASEA est en fait une sorte de compilateur qui utilise des fichiers sources écrits dans un langage simple pour produire un code C++ d'optimisation par AG qui peut ensuite être compilé et executé. L'idée est qu'EASEA doit être capable de traduire en C++ des fichiers assez simples pour être à l'avenir produits automatiquement par l'interface graphique. Pour l'instant cette dernière n'existe pas, et il faut donc écrire les fichiers initiaux. Cependant cette écriture se révèle très simple, et le code C++ généré se modifie facilement pour les besoins spécifiques de l'utilisateur, pourvu que ce dernier ait quelques notions de C ou C++. Il est en effet toujours plus facile de modifier un code existant que de l'écrire complètement. Le principe de fonctionnement d'EASEA est donné dans la figure (3.1).

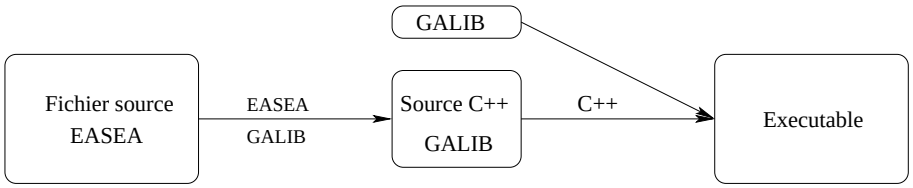

FIG. 3.1 – Fonctionnement d'EASEA

3.2.2 Utilisation d'EASEA avec le cas-test

L'objectif est bien sûr de réussir à l'aide d'EASEA l'optimisation du profil multiélément de la même manière, voire en mieux, qu'avec le code précédent.

Dans sa version actuelle (0.4), EASEA opère la traduction d'un fichier avec l'extension ".ez" en un fichier source C++. Ce code fait lui-même appel à une bibliothèque dédiée aux algorithmes génétiques, nommée GALib (pour la version GALib d'EASEA utilisée ici). Cette bibliothèque du domaine public contient de nombreux opérateurs et fonctions propres aux AG. La description du fichier utilisé par EASEA et du code généré est fournie dans l'annexe B. On se réfèrera aussi à [9].

L'adaptation du solveur Eulérien au code d'optimisation en C++ se révèle simple. Elle tient en deux modifications :

- Passer en paramètre un seul individu à la place de la population entière.
- Renvoyer l'adaptation calculée pour ce seul individu, prise égale à la portance.

La difficulté principale est l'interfaçage entre le Fortran 77 et le C++. Tous ces détails techniques sont donnés en annexe (cf. partie (B.3)).

3.2.3 Version parallèle d'EASEA

Face au problème posé par le temps de calcul, un prototype de version parallèle d'EASEA a été développé par Pierre Collet. Le principe est simple, il s'agit de procéder aux évaluations des individus sur des processus (et donc des processeurs) indépendants (cf. partie (2.4.3)). Cette implémentation est fondée sur l'utilisation des fonctionnalités MPI. Il est ensuite nécessaire d'adapter le solveur à la boucle d'optimisation parallèle. Il suffit pour cela de retirer les écritures sur fichiers faites dans la boucle principale d'optimisation.

Le code d'optimisation parallèle en C++ est fourni dans l'annexe C, et le code parallèle en C++ implémentant la stratégie de Nash est dans l'annexe D.

3.3 Résultats

3.3.1 Première expérience

Dans cette expérience, la portance C_L est maximisée selon les positions du bec et du volet, en utilisant le code en Fortran 77.

données

Pour le solveur :
- Mach = 0,12
- Incidence = 17,18 degrés

Pour l'algorithme génétique :
- Taille de la population = 30
- Nombre de générations = 40
- Probabilité de croisement = 0,9
- Probabilité de mutation = 0,05
- Selection par roulette
- Utilisation de la technique de transformation linéaire
- Stratégie élitiste

Résultats

Dans le tableau (3.1) sont résumées toutes les données concernant les paramètres de position.

Paramètre	**Intervalle**	**Valeur initiale**	**Valeur optimale**
δ_b	[28 ;34]	32,728	28,006
F_b	[0,019 ;0,032]	0,03125	0,01900
R_b	[-0,032 ;-0,02]	-0,03125	-0,02002
δ_v	[-37 ;-28]	-34,817	-36,851
F_v	[0,018 ;0,025]	0,02526	0,01898
R_v	[0,02 ;0,04]	0,04134	0,02002

TAB. 3.1 – Paramètres de position : données et résultats

La portance passe ainsi de 4,9078 à 5,1684.
La convergence du programme est résumée dans la figure (3.2).

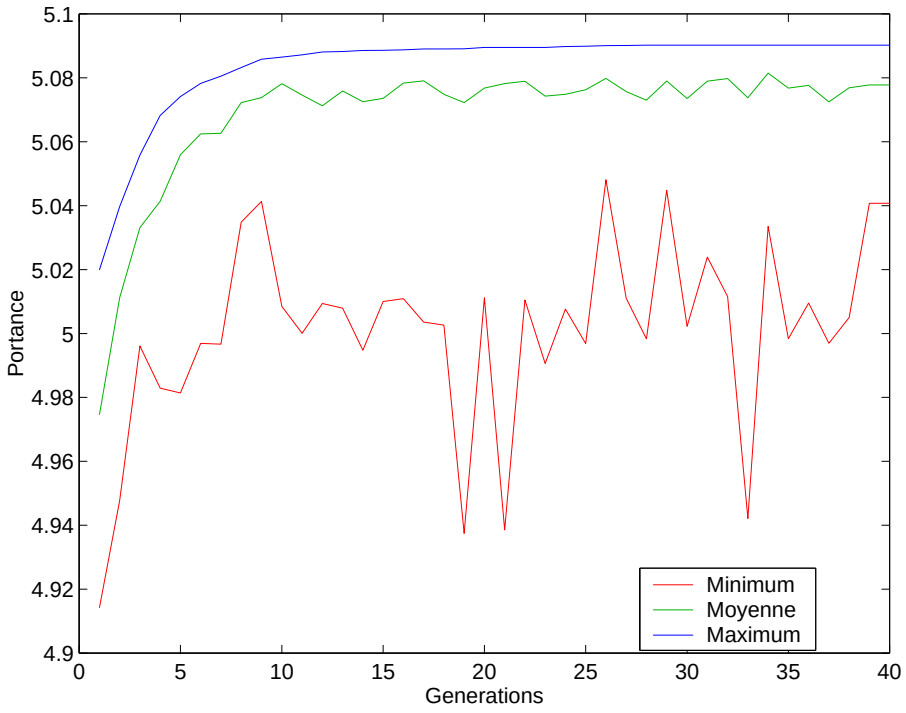

FIG. 3.2 – Convergence de l'algorithme génétique Fortran 77

On voit ici que le minimum et la moyenne de la portance progressent de manière très irrégulière, ce qui montre bien que l'AG explore bien le domaine de calcul, en testant des solutions mauvaises. Ceci limite les risques de convergence prématurée vers un optimum global. La stratégie élitiste (cf. partie (2.4.3)) permet d'obtenir la progression régulière observée pour le meilleur individu.

La durée d'exécution CPU est de 17 heures et cinq minutes, et le temps d'exécution est de 17 heures et 23 minutes, pour un total de 1200 évaluations d'individus, sur une station SUN sous Linux. Si on tient compte du temps occupé par l'initialisation et le calcul final, chaque évaluation dure environ 50 secondes. Il s'agit d'une durée très élevée, due à la complexité du calcul, et qui rend critique le besoin d'un code parallèle.

Les résultats concernant le problème physique seront donnés et commentés dans la partie (3.4).

3.3.2 Seconde expérience

Il s'agit de la même expérience que précédemment, mais en utilisant le code C++ généré par EASEA (cf. partie (3.2) et annexe B).

Données

Pour le solveur :

Cas de l'atterrissage :	Cas du décollage :
-Mach = 0,12	-Mach = 0,25
-Incidence = 17,18 degrés	-Incidence = 10 degrés

Pour l'algorithme génétique :

- Taille de la population = 30
- Nombre de générations = 40
- Probabilité de croisement = 0,9
- Probabilité de mutation = 0,05
- Sélection par échantillonnage stochastique sans remplacement
- Super-élitisme avec un pourcentage de remplacement de 90%

Dans le cas du décollage, la technique du changement d'échelle en puissance avec k = 1,0005 est utilisée.

Résultats

Dans le tableau (3.2) sont données les valeurs des paramètres de position qui optimisent la portance pour chacun des cas de vol. Les intervalles et les valeurs initiales pour ces paramètres sont les mêmes que précédemment (cf. tableau (3.1)).

Paramètre	**Valeur optimale cas du décollage**	**Valeur optimale cas de l'atterrissage**
δ_b	28,075	28,032
F_b	0,01904	0,01913
R_b	-0,02103	-0,02025
δ_v	-36,847	-36,841
F_v	0,01912	0,01913
R_v	0,02044	0,02016

TAB. 3.2 – Paramètres de position optimaux dans les deux cas de vol

Dans le cas du décollage, la portance passe de 4,3837 à 4,6528.
Dans le cas de l'atterrissage, la portance passe de 4,9078 à 5,1673.
La convergence du programme est donnée dans les figures (3.3) et (3.4).

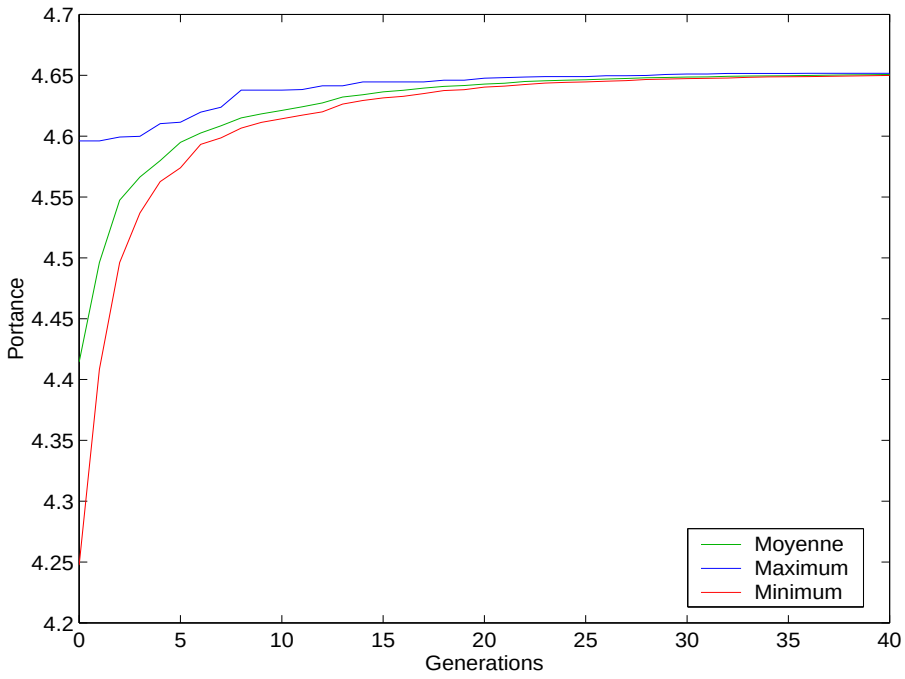

FIG. 3.3 – Convergence de l'algorithme génétique C++, cas du décollage

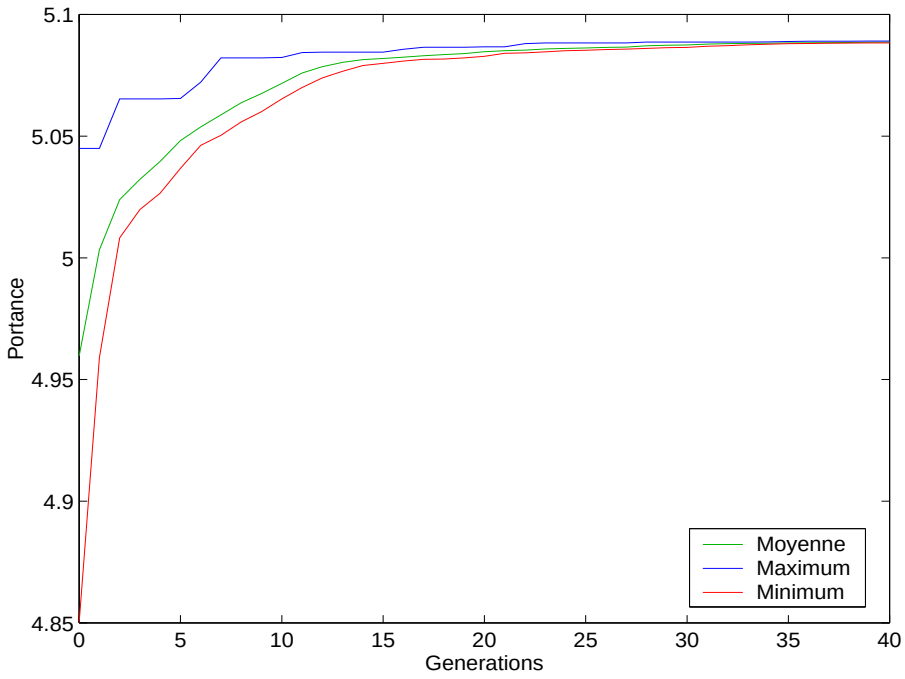

FIG. 3.4 – Convergence de l'algorithme génétique C++, cas de l'atterrissage

Par rapport au résultat précédent, la convergence est beaucoup plus régulière, y compris pour le minimum et la moyenne. Ceci est obtenu par l'emploi de la stratégie super-élitiste (cf. partie (2.4.3)), qui permet d'obtenir un algorithme plus stable.

La durée d'exécution CPU est cette fois-ci de 15 heures et 31 minutes et le temps d'exécution de 17 heures et 49 minutes pour la version séquentielle du code sur un Pentium II 450 sous Linux. Le gain en temps CPU est avant tout dû à l'emploi du super-élitisme. En effet à chaque génération il peut y avoir jusqu'à trois individus (10% d'une population de 30) qui ne sont pas remplacés, or le programme ne réévalue pas ces individus d'où un gain en nombre d'évaluations et en durée. La version parallèle s'exécute en deux heures avec 10 processus d'évaluation indépendants tournant chacun sur un Pentium III 500. Le gain en temps réel d'exécution est donc considérable, et ce avec un seul degré de parallélisme.

3.3.3 Troisième expérience : stratégie de Nash

La stratégie de Nash appliquée aux AG (cf. partie (2.4.4)) est testée et commentée ici. Deux joueurs sont définis, et ils optimisent tous deux la portance pour le même cas de vol, le premier selon les paramètres de position du volet, et le second selon ceux du bec. Ainsi :

$$1^{er}\text{joueur} : \max_{\delta_v, F_v, R_v} C_L$$
$$2^{eme}\text{joueur} : \max_{\delta_b, F_b, R_b} C_L$$

Données

Pour le solveur :

Cas de l'atterrissage :	Cas du décollage :
-Mach = 0,12	-Mach = 0,25
-Incidence = 17,18 degrés	-Incidence = 10 degrés

Pour les algorithmes génétiques (joueur 1 et 2) :

- Taille de la population = 30
- Nombre de générations = 40
- Probabilité de croisement = 0,9
- Probabilité de mutation = 0,05
- Selection par échantillonnage stochastique sans remplacement
- Super-élitisme avec un pourcentage de remplacement de 90%

La fréquence de Nash est de 1, c'est à dire que chaque joueur s'échangent les paramètres de leur meilleur individu entre chaque génération.

Dans le cas du décollage, la technique du changement d'échelle en puissance avec k = 1,0005 est utilisée (pour les deux joueurs).

On donne donc aux deux joueurs des paramètres identiques à ceux de l'AG simple, pour un cas de vol donné.

Résultats

Dans le tableau (3.3) sont données les valeurs des paramètres de position qui optimisent la portance pour chacun des cas de vol. Les intervalles et les

valeurs initiales pour ces paramètres sont les mêmes que précédemment (cf. tableau (3.1)).

Paramètre	**Valeur optimale cas du décollage**	**Valeur optimale cas de l'atterrissage**
δ_b	28,008	28,007
F_b	0,01902	0,01901
R_b	-0,02010	-0,02004
δ_v	-36,845	-36,844
F_v	0,01898	0,01905
R_v	0,02015	0,02001

TAB. 3.3 – Paramètres de position optimaux dans les deux cas de vol

Dans le cas du décollage, l'équilibre de Nash est de 4,6539.

Dans le cas de l'atterrissage, l'équilibre est de 5,1680.

Dans les figures (3.5) et (3.6) on compare la convergence des meilleurs individus des deux joueurs et du meilleur individu de l'AG simple, pour un cas de vol donné.

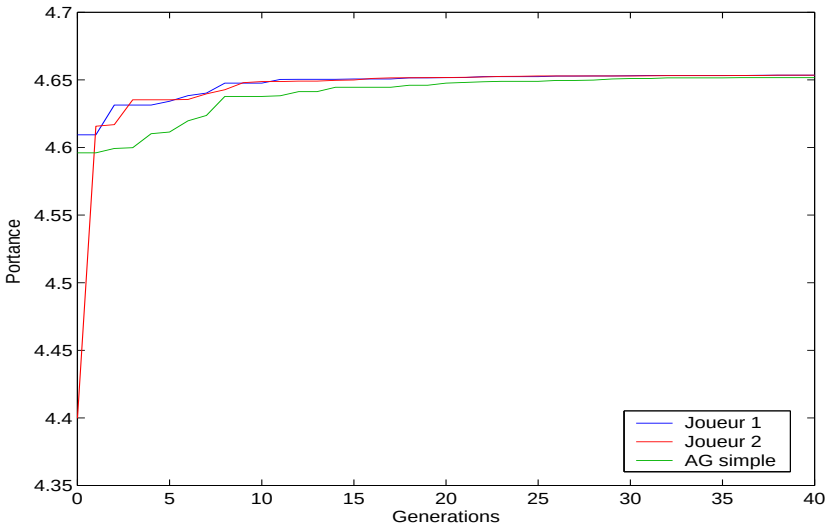

FIG. 3.5 – Convergence des meilleurs individus des joueurs 1 et 2 et de l'AG simple, cas du décollage

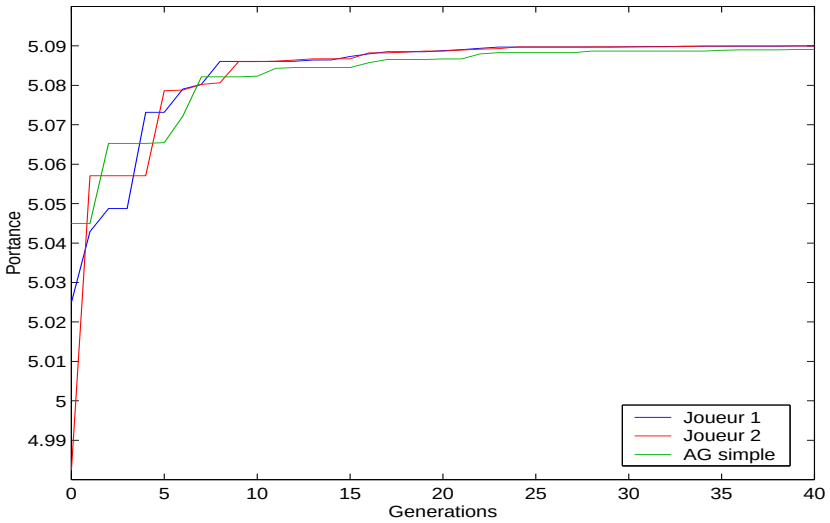

FIG. 3.6 – Convergence des meilleurs individus des joueurs 1 et 2 et de l'AG simple, cas de l'atterrissage

Conformément à ce qui était prévu, les deux joueurs convergent plus rapidement que l'AG simple dans les deux cas, et l'équilibre de Nash observé est dans chaque cas meilleur que l'optimum obtenu par l'AG simple.

3.4 Interprétation physique des résultats

3.4.1 Visualisation des champs de pression

Pour résumer, la portance d'une aile d'avion repose sur la différence de pression entre l'intrados et l'extrados. L'intrados est la face inférieure du profil et l'extrados est la face supérieure. Lorsque l'incidence est positive la courbure du profil est telle que la pression est plus élevée sur l'extrados que sur l'intrados, et l'aile est "soutenue" par cette différence, d'où la portance. Les champs de pression autour du profil dans les deux cas de vol sont donnés dans les figures suivantes ((3.7), (3.8), (3.9), (3.10)).

Les différences de pression apparaissent nettement. Cette dernière est sensiblement plus élevée sur l'intrados que sur l'extrados. Les différences les plus visibles apparaissent au niveau du bec. Dans les deux cas, la pression au niveau de l'intrados est plus élevée dans la solution optimale que dans la solution initiale. Les différences restent cependant subtiles, bien que la portance soit nettement plus élevée pour la solution optimale. La portance est donc très sensible aux changements, même limités, qui apparaissent dans l'écoulement.

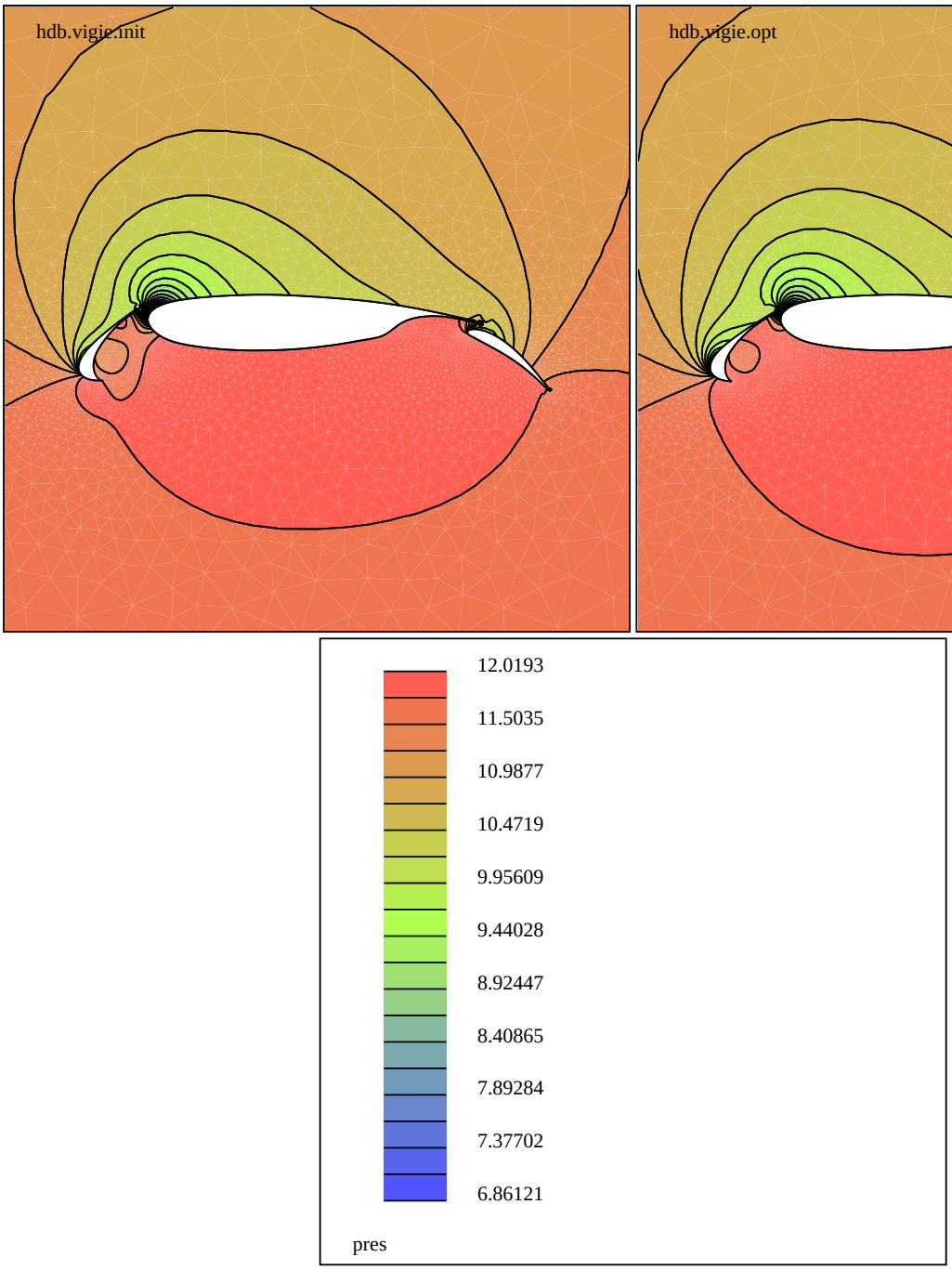

FIG. 3.7 – Champ de pressions autour du profil : cas du décollage

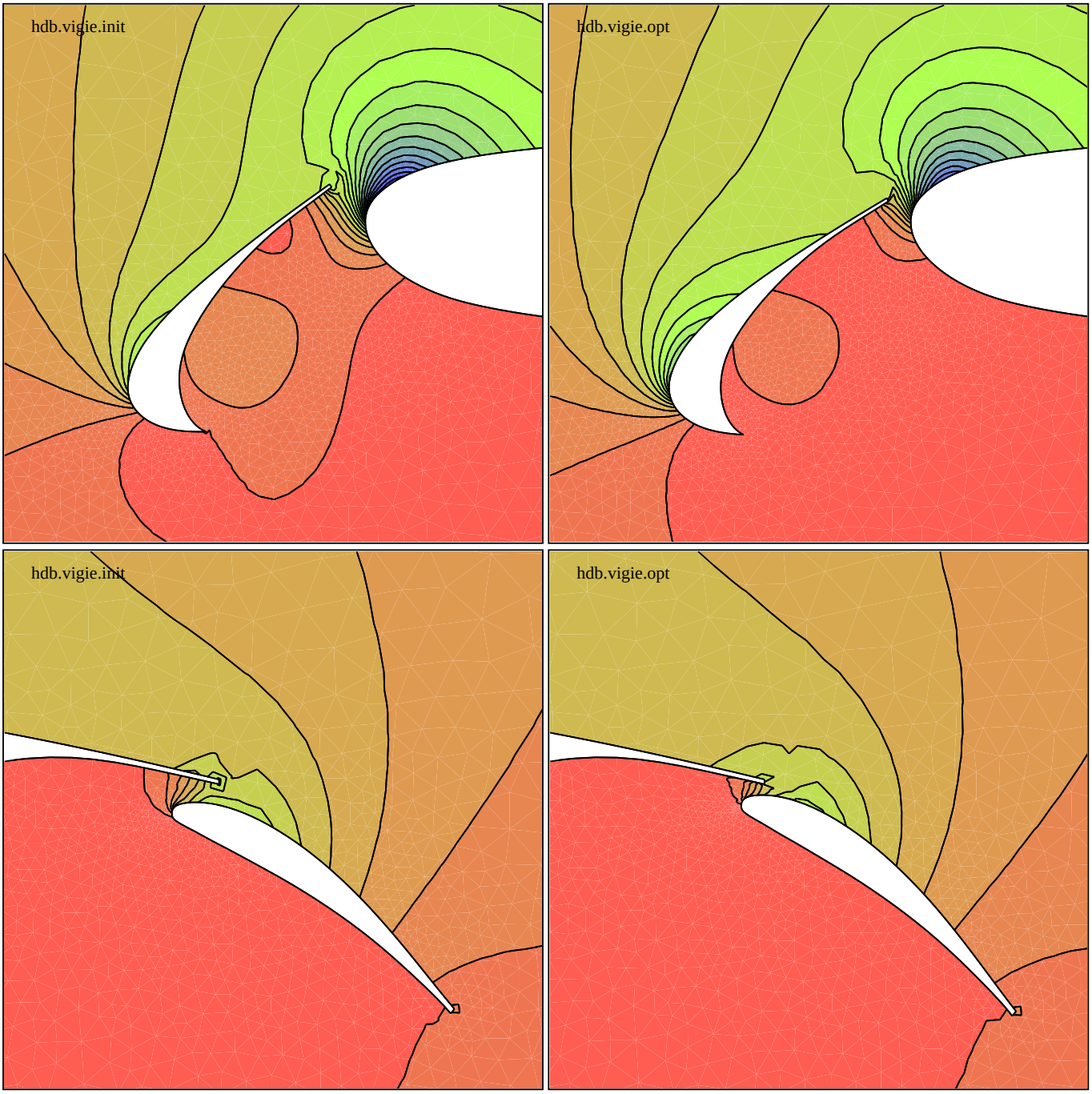

FIG. 3.8 – Champ de pressions près du bec et du volet : cas du décollage

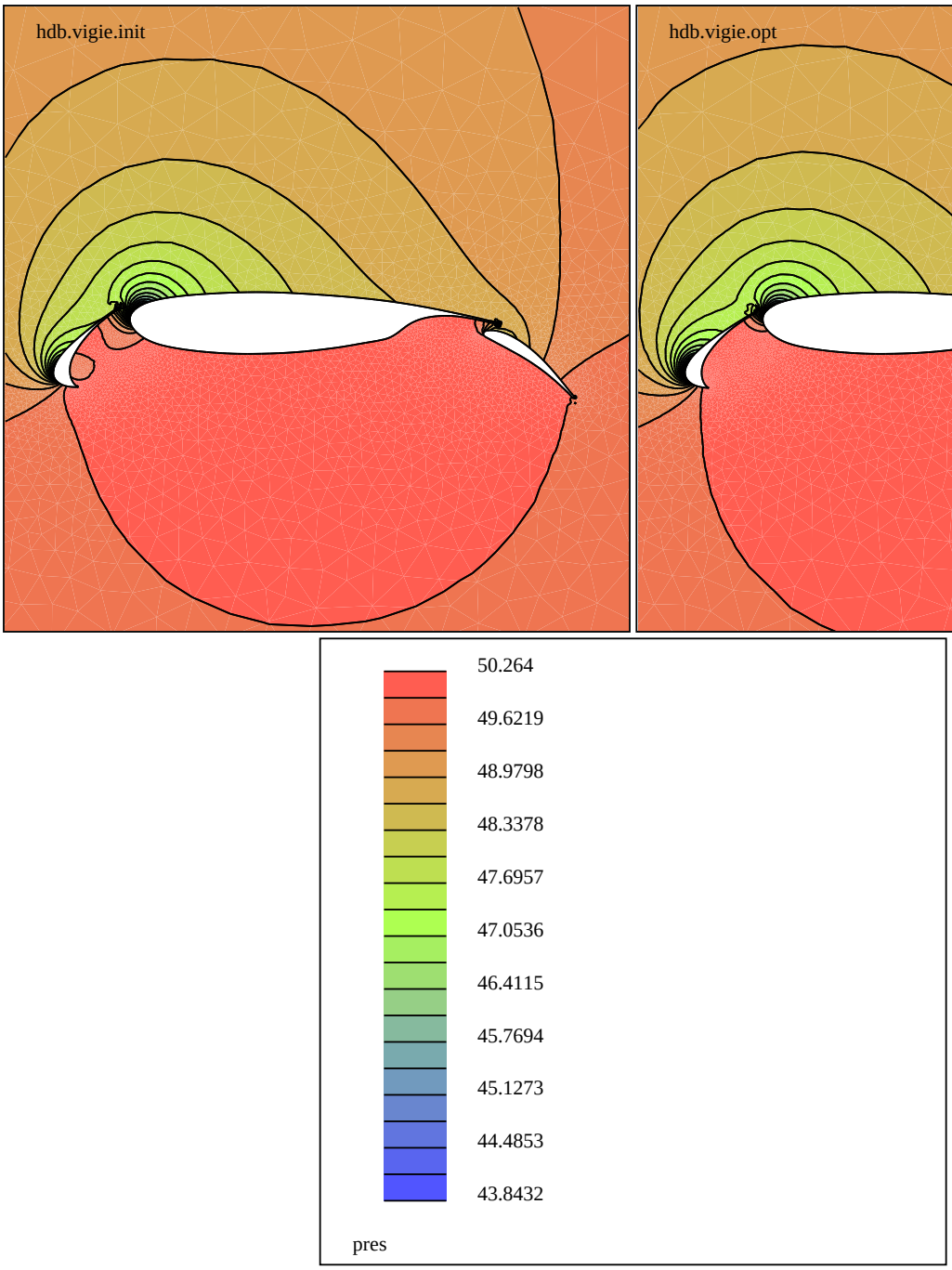

FIG. 3.9 – Champ de pressions autour du profil : cas de l'atterrissage

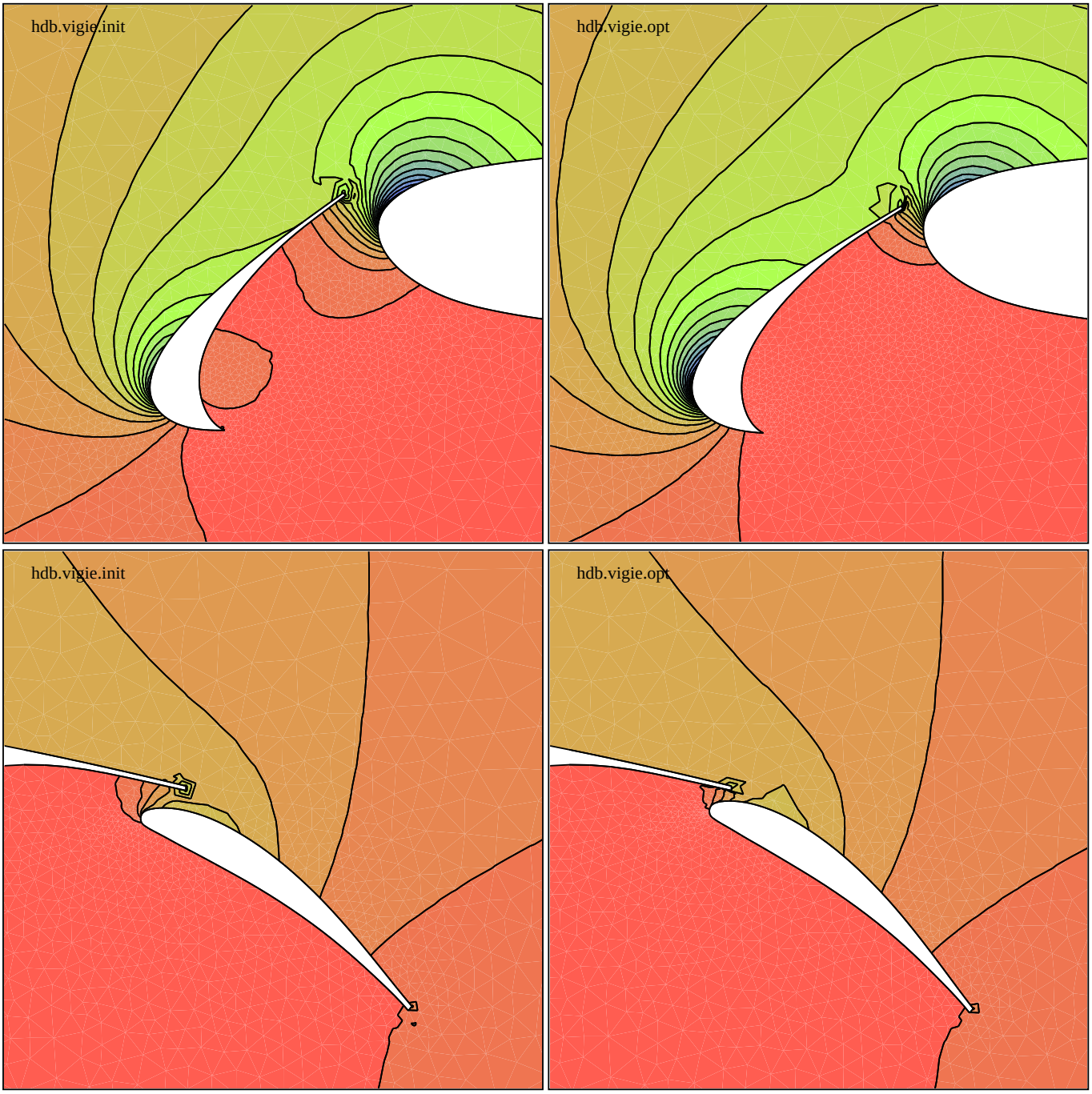

FIG. 3.10 – Champ de pressions près du bec et du volet : cas de l'atterrissage

3.4.2 Critiques

Le tableau (3.2) montre que dans les deux cas de vol, les paramètres de position optimaux sont pratiquement identiques. Les légères différences peuvent n'être dues qu'au caractère stochastique des AG. Dans les figures (3.11) et (3.12) sont visualisées les positions initiales et optimales du bec et du volet pour les deux cas de vol.

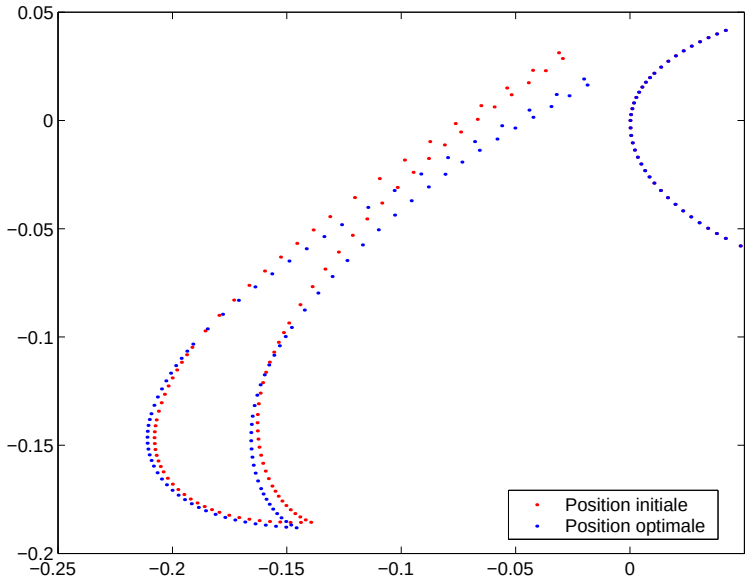

FIG. 3.11 – Position initiale et optimale du bec

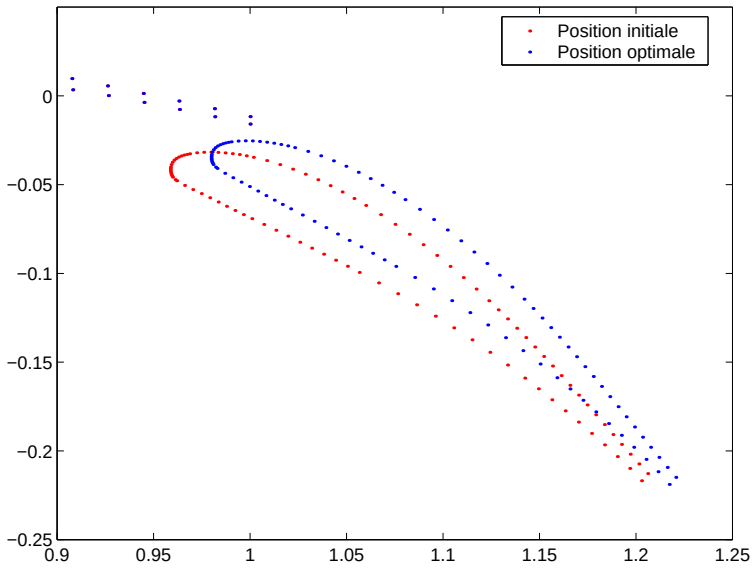

FIG. 3.12 – Position initiale et optimale du volet

Il semble assez curieux que pour deux cas de vol bien différents, la solution optimale soit la même. En fait, le bec est ouvert au minimum (δ_b est minimal) et le volet au maximum (δ_v maximal). Afin d'interpréter ce problème, une rencontre a été organisée avec Jian Feng Wang au Pôle Scientifique de Dassault Aviation. Il traite un problème très similaire, mais avec un solveur de type Stokes et couche limite qui prend donc en compte les phénomènes visqueux. Les différences apparaissent surtout au niveau du bec. J.F. Wang obtient dans tous les cas un volet ouvert (ou "braqué") au maximum, mais le bec s'ouvre un peu plus dans le cas de l'atterrissage.

Il semble que la différence soit surtout due à l'angle d'incidence, beaucoup plus élevé à l'atterrissage. En bref, la portance est créée quand l'écoulement passe sous l'aile et se trouve ralenti par la forme du profil. La différence de vitesse entre l'écoulement dans l'intrados (plus lent) et l'écoulement sur l'extrados (plus rapide) crée la différence de pression qui engendre la portance.

Si l'incidence est grande, le bec peut s'ouvrir plus, et l'écoulement dans l'intrados se trouve plus ralenti, d'où la portance plus élevée observée. De plus

la traînée est augmentée, ce qui est souhaitable puisqu'on cherche à décélérer à l'atterrissage.

Avec une incidence plus faible, si le bec est trop ouvert, ce dernier fait obstacle à l'écoulement, qui passe moins bien dans l'intrados, et une forte traînée est créée, d'où une chute de la portance. Le bec s'ouvre donc moins au décollage, ce qui évite par ailleurs d'avoir une trop forte traînée qui ralentirait l'avion.

Il n' y a pas de différences dans la position optimale du volet. En effet, une ouverture maximale favorise toujours la portance, car elle ralentit davantage l'écoulement dans l'intrados.

En conclusion, les solutions données par le solveur Eulérien sont physiquement insuffisantes. Les phénomènes visqueux jouent un rôle essentiel, surtout au niveau du bec. Il sera donc nécessaire dans l'étape suivante d'adopter un modèle de résolution plus proche de la réalité. L'idéal serait d'utiliser un solveur résolvant les équations de Navier-Stokes, et qui prendrait ainsi beaucoup mieux en compte les phénomènes visqueux. L'utilisation d'un maillage plus fin serait également souhaitable. On se heurte cependant au temps de calcul, déjà très élevé avec le modèle utilisé. Le choix des algorithmes génétiques est cependant judicieux et prometteur. Il s'agit d'une méthode novatrice dans un domaine tel que l'ingénierie aéronautique, très attachée à un certain déterminisme mathématique. Grâce à cette méthode, des optimisations prenant en compte des critères très variés, par exemple de furtivité, sont possibles.

Conclusion

Les algorithmes génétiques apparaissent comme une méthode d'optimisation très prometteuse. Appliqués au cas-test complexe d'un dispositif aérodynamique comme l'hyper-sustentateur, les résultats sont très encourageants. Le manque d'efficacité est compensé par la robustesse de l'algorithme, dont la durée peut être très largement réduite par le haut degré de parallélisme qu'il peut aisément atteindre. Ceci rend les AG très adaptés aux calculs scientifiques lourds. Des logiciels comme EASEA rendent par ailleurs la spécification de tels algorithmes accessibles aux non-spécialistes, et cela permet de plus de faire entrer en calcul scientifique des méthodes logicielles telles la programmation orientée objet. Les AG ont un autre intérêt : issus de l'optimisation en RO (Recherche Opérationnelle) et en simulation en variables discrètes, domaines ouverts à de telles méthodes semi-stochastiques, ils font le lien entre ces disciplines et l'ingénierie classique, très attachée aux méthodes déterministes, adaptées aux domaines continus. Le modèle physique adopté ici s'est révélé trop frustre, et l'utilisation de solveurs plus précis apparaît nécessaire. De nombreuses possibilités concernant les problèmes en aéronautique et dans d'autres disciplines restent à explorer par les algorithmes génétiques.

Bibliographie

[1] J.A. Desideri. *Modèles discrets et schémas itératifs, application aux algorithmes multigrilles et multidomaines*, pages 153-164. Editions Hermès, 1998.

[2] N. Marco and S.Lanteri. *A Two-Level Parallelization Strategy for GAs in Shape Optimum Design*, pages 6-12. Rapport de recherche n^o 3463, INRIA, 1998.

[3] P. Spiessens and B.Manderick. *A massively parallel Genetic Algorithm implementation and first analysis*. In Morgan Kaufmann, editor, 4^th *ICGA International Conference on Genetic Algorithms*, pages 279-285, San Mateo (CA), 1991.

[4] De Jong, K.A. *An analysis of the behavior of a class of genetic adaptive systems* (Doctoral dissertation, university of Michigan), Dissertation Abstracts International 36(10), 5140B (University Microfilms n^o 76-9381).

[5] D.E. Goldberg. *Algorithmes génétiques*, pages 36-40. Addison Wesley France, S.A., 1994.

[6] D.E. Goldberg. *Algorithmes génétiques*, pages 210-212. Addison Wesley France, S.A., 1994.

[7] D.E. Goldberg. *Algorithmes génétiques*, pages 87-90. Addison Wesley France, S.A., 1994.

[8] D.E. Goldberg. *Algorithmes génétiques*, pages 137-141. Addison Wesley France, S.A., 1994.

[9] P. Collet. *EASEA GALIB v0.4*. EVOLAB, INRIA, Ecole Polytechnique, ENSTA, Avril 2000.

Annexe A

Organigramme du code Fortran 77

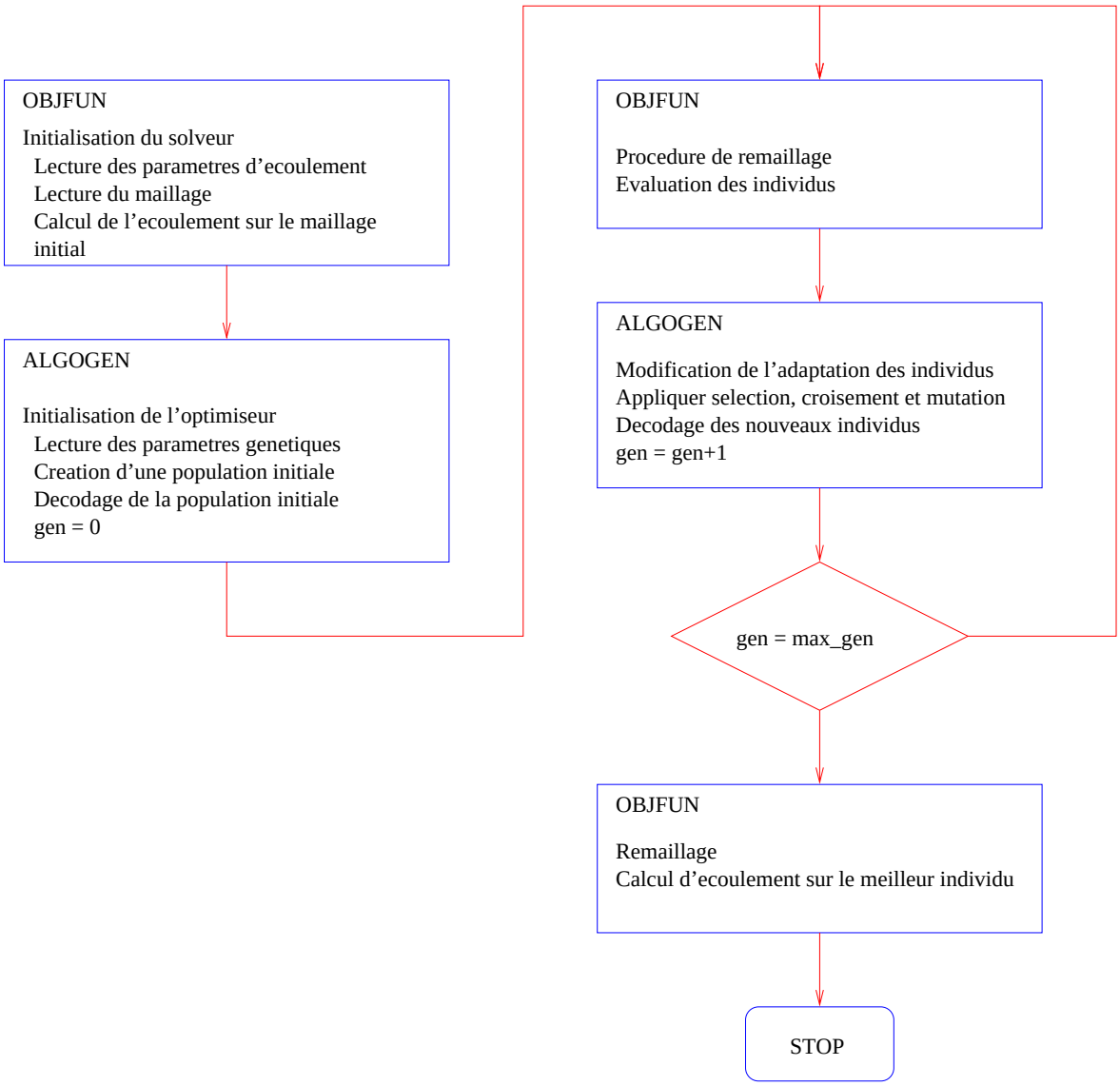

FIG. A.1 – Organigramme du code Fortran 77

Annexe B

EASEA

Le fonctionnement et l'utilisation d'EASEA sont détaillés ici, ainsi que la manière de "personnaliser" le code C++ généré afin d'utiliser certains opérateurs génétiques. On ne pourra se dispenser du manuel de Pierre Collet [9], et une visite sur la page web concernée sera utile :

`http://www-rocq.inria.fr/EVO-Lab/`

B.1 Le fichier EASEA

B.1.1 Contenu du fichier

Le contenu du fichier `MultiSeq.ez` utilisé afin de générer les codes C++ séquentiels d'optimisation par AG du dispositif hyper-sustentateur est donné *in extenso*. Les différences avec la version parallèle sont mineures.

```
User specific :
  #include <vector>
  int SIZE, phas_init, nopara, sizpop;
  double fit;
  vector<int>nbit(100);
  vector<double>coef1(100);
  vector<double>coef2(100);
  vector<double>coef3(100);
  extern "C" void objfun_();

Initialisation function:
/*********************************************************************
Entree des donnees necessaires a l'algorithme genetique :
elles sont intouchables. Elles sont probleme-dependant.
-------------------------------------------------------
nopara = nombre de parametres,
```

```
coef1(i) = min(i) et coef2(i) = max(i) (i=1,nopara) des parametres,
coef3(i) =  inverse de la resolution (2**nbit(i) - 1)/(max(i)-min(i)),
nbit(i) = nombre de bits determinant la chaine binaire pour chaque parametre,
SIZE = taille des chromosomes
**********************************************************************/
  double ddt;
  int i;
  std::ifstream entree("entree.dat0");  //Ouverture du fichier d'entree
  entree >> nopara;  //Lecture du nombre de parametres
  double* x=new double[nopara];
  double* precision=new double[nopara];
  objfun_(&phas_init,&nopara,&sizpop,xbest,&fit);
  for (i=0;i<nopara;i++)
  {
    entree >> coef1[i];  //Lecture des coef1(i)
  }
  for (i=0;i<nopara;i++)
  {
    entree >> coef2[i];  //Lecture des coef2(i)
  }
  for (i=0;i<nopara;i++)
  {
    entree >> precision[i];  //Lecture des precision(i)
  }
  entree.close();
  for(i=0;i<nopara;i++)
  {
    ddt=fabs(coef2[i]-coef1[i])/precision[i];
    nbit[i]=int(log(ddt)/log(2.0))+1;
    SIZE=SIZE+nbit[i];
    coef3[i]=(pow(2,nbit[i])-1)/fabs(coef2[i]-coef1[i]);
  }
  delete[] precision;
  delete[] x;

Genome { bool y[76]; }

Standard functions :

Genome::initialiser : // "initializer" is also accepted
  for (int i=0;i<SIZE;i++) Genome.y[i]=tossCoin(.5)?1:0;

Genome::crossover : // Must return the number of concerned children
  int GeneratedChildren=0;
  int CrossoverPosition=random(0,SIZE);
  if (&child1){
    for(int i=0;i<SIZE;i++)
      if (i<CrossoverPosition) child1.y[i]=parent1.y[i];
      else child1.y[i]=parent2.y[i];
    GeneratedChildren++;
  }
  if (&child2){
    for(int i=0;i<SIZE;i++)
      if (i<CrossoverPosition) child2.y[i]=parent2.y[i];
      else child2.y[i]=parent1.y[i];
    GeneratedChildren++;
  }
return GeneratedChildren;

Genome::mutator : // Must return the number of mutations
  int NbMut=0;
  for (int i=0;i<SIZE;i++)
    if (tossCoin(PMut)){
```

```
      NbMut++;
      Genome.y[i]=Genome.y[i]?0:1;
    }
  return NbMut;

Genome::evaluator : // Must return the score as a double
/*************************************************************************
On decode une chaine binaire en la valeur reelle des parametres
x (reel) = xmin + (valeur de la chaine binaire)*(xmax-xmin)/(2**lchrom-1)
valeur de la chaine binaire = Somme (de i = 0 a lchrom-1) ai * 2**i
ou ai = 0 ou 1, ensuite on appelle le solveur qui renvoie la fitness
*************************************************************************/
  double *x=new double[nopara];
  int sbit,ebit,powerof2;
  double accum;
  sbit=0;
  ebit=0;
  for (int i=0;i<nopara;i++)
  {
    ebit+=nbit[i];
    accum=0;
    powerof2=1;
    for (int j=sbit;j<ebit;j++)
    {
      accum=accum+Genome.y[j]*powerof2;
      powerof2=powerof2*2;
    }
    x[i]=accum/coef3[i];
    x[i]=x[i]+coef1[i];
    sbit=ebit;
  }
  phas_init=1;
  objfun_(&phas_init,&nopara,&sizpop,x,&fit);  //Appel du solveur
  delete[] x;
  return fit;


Run parameters :
  Population size : 30         // PSize
  Number of generations : 40 // NbGen
  Mutation probability : 0.05    // PMut
  Crossover probability : 0.9     // PCross
  Replacement percentage : 90%  // ReplPerc
  Genetic engine : SteadyState

End of genome file.
```

La syntaxe ressemble beaucoup à du C++, en bien plus simple. La compréhension est en fait intuitive avec de simples notions de C.

B.1.2 Analyse du code

La partie `User specific`

Elle peut contenir tout code C++ spécifique à l'utilisateur. Son contenu sera intégralement recopié dans le code généré. C'est pour cela que sa syntaxe

doit être conforme au C++ ou au C. Dans le cas présent une inclusion de librairie, des déclarations de variables globales et la déclaration du solveur.

La fonction `Initialisation function`

Cette fonction est simplement recopiée dans le code C++. Sa syntaxe doit donc être conforme à ce langage (ou au C). Elle est appelée avant toute autre dans le `main` du code C++. Elle doit retourner `void`. l'utilisateur peut avec cette fonction procéder à toutes les initialisations qu'il désire. Ici, elle sert à lire les informations nécessaires au codage/décodage des génomes, et elle procède au premier appel du solveur.

Déclaration du génome

```
Genome { bool y[76]; }
```

La déclaration du génome ressemble à du C++ simplifié. Ici le génome est simplement un tableau de booléen. Il est nécessaire de spécifier la taille qu'il doit faire, il n'est pas possible de le faire dynamiquement.

Le chromosome proprement dit (c'est à dire le tableau de booléens) constitue le membre de la classe `Genome`. Dans toutes les méthodes décrites ci-dessous, il est nécessaire pour accéder aux éléments de ce tableau de passer par la variable `Genome`. Par exemple le quatrième élément du tableau est accessible par `Genome.y[3]`.

La fonction `initialiser`

Son rôle est d'initialiser les chromosomes de la toute première génération.

La fonction `tossCoin` renvoie un booléen. Sa probabilité d'être vraie est égale à la valeur de son argument. Ainsi `tossCoin(0.5)` renvoie 1 avec 50% de chances. En définitive, le génome est initialisé de façon aléatoire.

La fonction `crossover`

Cette fonction est appelée selon la probabilité de croisement. Telle qu'elle est écrite, elle applique le croisement décrit dans la partie (2.1.4). La fonction

doit renvoyer la variable GeneratedChildren qui désigne le nombre de genomes issus du croisement (ici deux).

La fonction `mutator`

Cette fonction applique sur un génome la mutation telle qu'elle est décrite dans la partie (2.1.4). Chaque bit est permuté avec une probabilité `PMut`. La fonction doit renvoyer la variable `NbMut` qui désigne le nombre de bits mutés.

la fonction `evaluator`

L'utilisateur peut spécifier cette fonction de n'importe quelle manière, mais la valeur d'adaptation renvoyée doit être *positive*. Ici, la fonction procède au décodage du génome avant d'appeler le solveur en Fortran 77.

La partie `Run parameters`

Dans cette partie du code sont spécifiés les paramètres de l'AG. Seuls six paramètres sont pour le moment reconnus par EASEA.

La ligne `Replacement percentage` désigne la part de la population qui doit être remplacée d'une génération à l'autre. Ce paramètre n'est utile que si le type d'AG est `SteadyState`. Il s'agit du sixième paramètre. Il faut spécifier `SteadyState` si on veut utiliser la stratégie de super-élitisme (cf. partie (2.4.3)). Sinon, il faut remplacer `SteadyState` par `Simple`.

B.2 Génération et modification du code

B.2.1 Génération du code

Trois éléments sont nécessaires pour générer le code C++ à partir du fichier EASEA. La première est l'exécutable EASEA lui-même. Ce dernier a besoin du fichier nommé `GALib.tpl` et bien sûr du fichier EASEA lui-même.

Dans le cas du programme "MultiSeq", la génération du code est très simple. On obtient la sortie suivante après avoir tapé `easea MultiSeq` :

```
                  +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
                  |             E A S E A                 |         V0.4
                  +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

Inserting User-Specific section.
Inserting user specific functions and declarations.
Inserting initialisation function.

Genome declaration analysis :

    y array declared (76 bytes)
                  ---------------------------------------

Generation of the C++ genetic algorithm source file for MultiSeq : MultiSeq.cpp

Inserting User-Classes.
Creating   default genome constructor.
Creating   default genome constructor.
Creating  default destructor.
Inserting genome.
Creating  default copy constructor.
Creating  default diversity test.
Creating  default genome comparator.
Creating  default read command.
Creating  default write command.
Inserting standard functions.
Inserting user constructor from .ez file.
Inserting user crossover from .ez file.
Inserting user mutator from .ez file.
Inserting user evaluator from .ez file.

CONGRATULATIONS !!
MultiSeq.cpp file generation succeeded.
If you are running EASEA under UNIX, you may now compile your file with:

g++ MultiSeq.cpp -o MultiSeq -lga

Don't forget to compile for speed when you are finished with debugging.
Have a nice compile time.
```

Le code du fichier EASEA est pratiquement recopié dans le code C++ généré. Si ce fichier contient des erreurs, elles sont signalées par le compilateur C++.

Attention ! Il existe deux versions du fichier "GALib.tpl" : une parallèle et une séquentielle. Selon que l'on voudra générer un code séquentiel ou parallèle, il faudra utiliser l'une ou l'autre, mais le nom du fichier doit toujours être "GALib.tpl".

B.2.2 Code généré et modifications

Le code généré à partir du fichier précédent est donné ci-après. Il s'agit donc du code séquentiel d'optimisation par AG de la position du bec et du volet de l'hyper-sustentateur.

```
//*****************************************
//
//  MultiSeq.cpp
//
//  C++ file generated by EASEA-GALIB v0.4
//
//*****************************************

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <iostream.h>
#include <fstream.h>
#include <time.h>
#include <ga/ga.h>

int MultiSeq_EVALUATED_GENERATIONS=0;
int MultiSeq_NB_EVALUATIONS=0;
clock_t   MultiSeq_START, MultiSeq_FINISH;
double MultiSeq_LET=0; // Last Elapsed Time
double MultiSeq_LERT=1; // Last Evaluated Remaining Time
float ReplPerc=0;
int NbGen, PSize;

// User Specific
// User Functions and Declarations

#include <vector>
  int SIZE, phas_init, nopara, sizpop;
  double fit;
  vector<int>nbit(100);
  vector<double>coef1(100);
  vector<double>coef2(100);
  vector<double>coef3(100);
  extern "C" void objfun_();

// Initialisation function

void EASEAInitFunction(int argc, char *argv[]){
/*********************************************************************
Entree des donnees necessaires a l'algorithme genetique :
elles sont intouchables. Elles sont probleme-dependant.
-----------------------------------------------------
nopara = nombre de parametres,
coef1(i) = min(i) et coef2(i) = max(i) (i=1,nopara) des parametres,
coef3(i) =  inverse de la resolution (2**nbit(i) - 1)/(max(i)-min(i)),
nbit(i) = nombre de bits determinant la chaine binaire pour chaque parametre,
SIZE = taille des chromosomes
*********************************************************************/
  double ddt;
  int i;
  std::ifstream entree("entree.dat0");  //Ouverture du fichier d'entree
  entree >> nopara;  //Lecture du nombre de parametres
  double* x=new double[nopara];
  double* precision=new double[nopara];
```

```
  objfun_(&phas_init,&nopara,&sizpop,xbest,&fit);
  for (i=0;i<nopara;i++)
  {
    entree >> coef1[i];  //Lecture des coef1(i)
  }
  for (i=0;i<nopara;i++)
  {
    entree >> coef2[i];  //Lecture des coef2(i)
  }
  for (i=0;i<nopara;i++)
  {
    entree >> precision[i];  //Lecture des precision(i)
  }
  entree.close();
  for(i=0;i<nopara;i++)
  {
    ddt=fabs(coef2[i]-coef1[i])/precision[i];
    nbit[i]=int(log(ddt)/log(2.0))+1;
    SIZE=SIZE+nbit[i];
    coef3[i]=(pow(2,nbit[i])-1)/fabs(coef2[i]-coef1[i]);
  }
  delete[] precision;
  delete[] x;

}

// User Classes


inline int random(int b1=0, int b2=1){
  return GARandomInt(b1,b2);
}
inline double random(double b1=0, double b2=1){
  return GARandomDouble(b1,b2);
}
inline float random(float b1=0, float b2=1){
  return GARandomFloat(b1,b2);
}

// User Genome
class MultiSeqGenome : public GAGenome {
// Default methods for class MultiSeqGenome
public:
  GADefineIdentity("MultiSeqGenome", 251);
  static void Initializer(GAGenome&);
  static int Mutator(GAGenome&, float);
  static float Comparator(const GAGenome&, const GAGenome&);
  static float Evaluator(GAGenome&);
  static int Crossover(const GAGenome&, const GAGenome&, GAGenome*, GAGenome*);
public:
  MultiSeqGenome::MultiSeqGenome() :GAGenome(Initializer, Mutator, Comparator){
    evaluator(Evaluator); crossover(Crossover);
  }
  MultiSeqGenome(const MultiSeqGenome & orig) {
    copy(orig);
  }
  ~MultiSeqGenome() {
// Destructing pointers
  }
  MultiSeqGenome& operator=(const MultiSeqGenome &);
  virtual GAGenome *clone(GAGenome::CloneMethod) const ;
  virtual void copy(const GAGenome & c);
  virtual int equal(const GAGenome& g) const;
```

```
  virtual int read(istream & is);
  virtual int write(ostream & os) const ;

// Class members
  bool y[76];
};

MultiSeqGenome& MultiSeqGenome::operator=(const MultiSeqGenome & arg){
  copy(arg);
  return *this;
}

void MultiSeqGenome::copy(const GAGenome& g) {
  if(&g != this){
    GAGenome::copy(g);    // copy the base class part
    MultiSeqGenome & genome = (MultiSeqGenome &)g;
// Memberwise copy
    {for(int EASEA_Ndx=0; EASEA_Ndx<76; EASEA_Ndx++)
       y[EASEA_Ndx]=genome.y[EASEA_Ndx];}
  }
}

GAGenome*MultiSeqGenome::clone(GAGenome::CloneMethod) const {
  return new MultiSeqGenome(*this);
}

int MultiSeqGenome::equal(const GAGenome& g) const {
  MultiSeqGenome& genome = (MultiSeqGenome&)g;
// Default diversity test (required by GALib)
  {for(int EASEA_Ndx=0; EASEA_Ndx<76; EASEA_Ndx++)
     if (y[EASEA_Ndx]!=genome.y[EASEA_Ndx]) return 0;}
  return 1;
}

float MultiSeqGenome::Comparator(const GAGenome& a, const GAGenome& b) {
  MultiSeqGenome& sis = (MultiSeqGenome &)a;
  MultiSeqGenome& bro = (MultiSeqGenome &)b;
  int diff = 0;
// Default genome comparator (required by GALib)
  {for(int EASEA_Ndx=0; EASEA_Ndx<76; EASEA_Ndx++)
     if (sis.y[EASEA_Ndx]!=bro.y[EASEA_Ndx]) diff++;}
  return (float)diff;
}

int MultiSeqGenome::read(istream & is) {
// Default read command (required by GALib)
  return is.fail() ? 1 : 0;
}

int MultiSeqGenome::write(ostream & os) const {
// Default write command (required by GALib)
  {os << "Array y : ";
   for(int EASEA_Ndx=0; EASEA_Ndx<76; EASEA_Ndx++)
     os << "[" << EASEA_Ndx << "]:" << y[EASEA_Ndx] << "\t";}
  os << "\n";
  return os.fail() ? 1 : 0;
}


// Standard Functions

void MultiSeqGenome::Initializer(GAGenome& g) {
  MultiSeqGenome & genome = (MultiSeqGenome &)g;
```

```
 // "initializer" is also accepted
  for (int i=0;i<SIZE;i++) genome.y[i]=GAFlipCoin(.5)?1:0;

genome._evaluated=gaFalse;
}

int MultiSeqGenome::Crossover(const GAGenome& a, const GAGenome& b, GAGenome* c, GAGenome* d) {
  MultiSeqGenome& mom = (MultiSeqGenome &)a;
  MultiSeqGenome& dad = (MultiSeqGenome &)b;
  MultiSeqGenome& sis = (MultiSeqGenome &)*c;
  MultiSeqGenome& bro = (MultiSeqGenome &)*d;
  if(&bro) bro._evaluated=gaFalse;
  if(&sis) sis._evaluated=gaFalse;
 // Must return the number of concerned children
  int GeneratedChildren=0;
  int CrossoverPosition=random(0,SIZE);
  if (&bro){
    for(int i=0;i<SIZE;i++)
      if (i<CrossoverPosition) bro.y[i]=dad.y[i];
      else bro.y[i]=mom.y[i];
    GeneratedChildren++;
  }
  if (&sis){
    for(int i=0;i<SIZE;i++)
      if (i<CrossoverPosition) sis.y[i]=mom.y[i];
      else sis.y[i]=dad.y[i];
    GeneratedChildren++;
  }
return GeneratedChildren;

}

int MultiSeqGenome::Mutator(GAGenome& g, float pmut) {
  MultiSeqGenome & genome = (MultiSeqGenome &)g;
  genome._evaluated=gaFalse;
 // Must return the number of mutations
  int NbMut=0;
  for (int i=0;i<SIZE;i++)
    if (GAFlipCoin(pmut)){
       NbMut++;
       genome.y[i]=genome.y[i]?0:1;
    }
  return NbMut;
}

float MultiSeqGenome::Evaluator(GAGenome & c) {
  MultiSeqGenome & genome = (MultiSeqGenome &)c;
  MultiSeq_NB_EVALUATIONS++;
 // Must return (float) the score as a double
/*************************************************************************
On decode une chaine binaire en la valeur reelle des parametres
x (reel) = xmin + (valeur de la chaine binaire)*(xmax-xmin)/(2**lchrom-1)
valeur de la chaine binaire = Somme (de i = 0 a lchrom-1) ai * 2**i
ou ai = 0 ou 1, ensuite on appelle le solveur qui renvoie la fitness
*************************************************************************/
  double *x=new double[nopara];
  int sbit,ebit,powerof2;
  double accum;
  sbit=0;
  ebit=0;
  for (int i=0;i<nopara;i++)
  {
    ebit+=nbit[i];
```

```
    accum=0;
    powerof2=1;
    for (int j=sbit;j<ebit;j++)
    {
      accum=accum+genome.y[j]*powerof2;
      powerof2=powerof2*2;
    }
    x[i]=accum/coef3[i];
    x[i]=x[i]+coef1[i];
    sbit=ebit;
  }
  phas_init=1;
  objfun_(&phas_init,&nopara,&sizpop,x,&fit);  //Appel du solveur
  delete[] x;
  return (float) fit;


}



int main(int argc, char *argv[]){
  FILE *MultiSeq_PRM;
  int i;

  GARandomSeed(0);

// Checks whether we've been given a seed to use (for testing purposes).
  for(i=1; i<argc; i++)
    if(strcmp(argv[i++],"seed") == 0)
      GARandomSeed((unsigned int)atoi(argv[i]));

  EASEAInitFunction(argc, argv);

  MultiSeqGenome genome;
  GASteadyStateGA ga(genome);
  ga.populationSize(3); // how many individuals in the population
  ga.nGenerations(4);    // number of generations to evolve
  ga.pReplacement((float)90.000000/100);    // percentage of the population to be replaced
  ga.pMutation((float)0.050000);    // likelihood of mutating new offspring
  ga.pCrossover((float)0.900000);   // likelihood of crossing over parents
  if (MultiSeq_PRM=fopen("MultiSeq.prm","r")){
    fclose(MultiSeq_PRM);
    ga.parameters("MultiSeq.prm"); // gets GALib parameters from the MultiSeq.prm file
  }
  ga.parameters(argc, argv); // gets GALib parameters from the command line
  PSize=ga.populationSize();
  NbGen=ga.nGenerations();
  ReplPerc=ga.pReplacement()*100;

  genome.initialize();
  cout << "Score of a generation 0 genome: " << genome.Evaluator(genome) << "\n";
  cout << "Contents of the genome:\n" << genome << endl;

  MultiSeq_START=clock();
  ga.initialize();
  fprintf(stderr,"Estimated Remaining Time:           ");
  do{
    ga.step();
    MultiSeq_EVALUATED_GENERATIONS++;
    MultiSeq_FINISH=clock();
    if ((double)(MultiSeq_FINISH-MultiSeq_START)-MultiSeq_LET > CLOCKS_PER_SEC){
      for (i=0;i<log10(MultiSeq_LERT);i++) fprintf(stderr,"\b");
```

```
      MultiSeq_LET=(double)(MultiSeq_FINISH-MultiSeq_START) ;
      MultiSeq_LERT=(MultiSeq_LET*(ga.nGenerations()-MultiSeq_EVALUATED_GENERATIONS)/MultiSeq_EVALUATED_GENERATIONS);
      MultiSeq_LERT/=CLOCKS_PER_SEC;
      fprintf(stderr,"\b\b\b\b\b\b\b\b\b\b%d seconds  ",(int)MultiSeq_LERT);
    }
  } while(!ga.done());
  cout << "\nBest genome score: " << (ga.statistics().bestIndividual()).evaluate() << endl;
  cout << "Contents of the genome:\n" << ga.statistics().bestIndividual();
  cout << "Elapsed time: " << (double)( MultiSeq_FINISH-MultiSeq_START) /CLOCKS_PER_SEC<< " seconds for ";
  cout << MultiSeq_NB_EVALUATIONS << " evaluations.\n";
  cout << "Main parameters: PSize=" << PSize << ", NbGen=" << NbGen <<", PMut=" << ga.pMutation();
  cout<< ", PCross=" << ga.pCrossover();
  cout << ", ReplPerc=" << ReplPerc;
  cout << "\n";
  exit(0);
  return 0;
}

// If your compiler does not do automatic instantiation (e.g. g++ 2.6.8),
// then define the NO_AUTO_INST directive.  This will force the instantiation
// of the template classes that we use.  For some compilers (e.g. metrowerks)
// this must come after any specializations or you'll get 'multiply-defined'
// errors when you compile.
#ifdef NO_AUTO_INST
#include "GAList.cpp"
#include "GAListGe.cpp"
#if defined(__GNUG__)
template class GAList<int>;
template class GAListGenome<int>;
#else
GAList<int>;
GAListGenome<int>;
#endif
#endif


// That's all folks !
```

On reconnaît très bien le contenu du fichier EASEA dans ce code. De base ce code est assez frustre car il ne premet pas d'obtenir de statistiques sur l'évaluation de la population, la méthode de sélection est la roulette, et on n'a pas de procédure de modification de l'adaptation.

B.2.3 Modifications utiles

L'utilisateur, avec des connaissances de C ou de C++, pourra facilement ajouter les fonctions qu'il juge utile. Mais l'utilisation des fonctionnalités de GALib se révèle très opportune, bien que nécessitant cette fois-ci une compréhension du C++.

Modification des paramètres de l'AG

Dans le code C++ l'AG est un objet déclaré ainsi :

```
GASteadyStateGA ga(genome);
```

L'objet `ga` possède des méthodes pour changer la taille de la population, le nombre de générations, les probabilités de croisement et de mutation, ainsi que le pourcentage de remplacement. Il suffit donc de modifier les lignes suivantes du code pour procéder aux changements :

Changer la taille de la population :

```
ga.populationSize(PSize); //PSize est la taille de la population voulue
```

Changer le nombre de générations :

```
ga.nGenerations(NbGen); //NbGen est le nombre de generations
```

Changer le pourcentage de remplacement :

```
ga.pReplacement((float)PRepl.000000/100); //Pour avoir PRepl% de la population
                                          //remplacee entre deux generations
```

Changer les probabilités de croisement et de mutation :

```
ga.pMutation((float)PMut); //PMut : probabilite de mutation
ga.pCrossover((float)Pcross); //Pcross : probabilite de croisement
```

Modifier le schéma de sélection

GALIB fournit des procédures de sélection variées : tournoi, roulette, échantillonnage stochastique sans remplacement, etc...

`ga` désigne l'AG. Il possède une méthode `selector` qui lui permet de changer de méthode de sélection. Il faut pour cela déclarer cette méthode, qui est elle aussi un objet. GALIB fournit plusieurs classes pour déclarer de tels objets. En définitive pour utiliser par exemple l'échantillonnage stochastique sans remplacement il faut ajouter au code les lignes suivantes :

```
GASRSSelector SRSSelector; //Tout autre nom que SRSSelector est
                           //recevable
ga.selector(SRSSelector);
```

En deux lignes, on obtient ainsi une sélection non triviale.

Autres modifications

De la même manière, utiliser la technique du changement d'échelle en puissance on ajoute :

```
GAPowerLawScaling PowerLawScaling; //Encore une fois, tout autre nom
                                   //est acceptable
ga.scaling(PowerLawScaling);
```

Et pour obtenir dans un fichier "maxavgmin.dat" la moyenne, le maximum et le minimum de l'adaptation de la population à chaque génération il faut ajouter :

```
ga.scoreFilename("maxavgmin.dat");
ga.selectScores(GAStatistics::AllScores);
ga.scoreFrequency(1);
ga.flushFrequency(1);
```

Il est nécessaire après chaque changement de recompiler le code. Bien entendu GALib offre de bien plus nombreuses fonctionnalités, et l'on se reportera à la documentation HTML.

B.3 Compilation du code généré

On s'intéresse avant tout ici au problème d'interfaçage entre le solveur en Fortran 77 et l'optimiseur en C++.

B.3.1 Appel du solveur

Il est d'abord nécessaire de le déclarer ainsi :

```
extern "C" void objfun_();
```

Important : le nom de la procédure doit être en minuscules et suivi d'un "trait bas" *dans la déclaration uniquement.* Mais le nom de la procédure Fortran 77 *ne comporte pas ce caractère.*

L'appel se fait de la manière suivante :

```
objfun_(&phas_init,&nopara,&sizpop,xbest,&fit);
```

Tous les paramètres *sauf les tableaux* doivent être passés par référence.

B.3.2 Compilation du code

Pour la version séquentielle, g77 (pour le Fortran) et g++ (pour le C++) sont utilisés Pour la version parallèle, ce sont respectivements mpiF77 et mpiCC pour le Fortran et le C++ qui sont utilisés.

Il est nécessaire pour compiler `MultiSeq.cpp` de procéder à des compilations et une édition des liens séparées. Pour simplifier, on fera comme si le solveur `Objfun` ne fait pas d'appel à d'autres routines.

Compilation :

```
g77 -c Objfun.f
g++ -c MultiSeq.cpp -I/usr/local/galib/include
```

Cette première étape génère les objets `MultiSeq.o` et `Objfun.o`.

Pour l'édition des liens, il est nécessaire de faire appel à la librairie GALib et à la librairie standard du Fortran 77, car l'édition des liens se fait avec g++ ou mpiCC :

```
g++ -o MultiSeq MultiSeq.o Objfun.o -L/usr/local/galib/lib -lga
-L/usr/local/gcc/lib/gcc-lib/i686-pc-linux-gnu/2.95.2 -lg2c
```

Il convient de préciser que l'exécutable généré ainsi n'est (pour l'instant) utilisable que sur un PC Linux (ou sur un cluster de PC).

Annexe C

Code d'optimisation parallèle

```
//****************************************
//
//  MultiPar.cpp
//
//  C++ file generated by EASEA-GALIB v0.4
//
//****************************************

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <iostream.h>
#include <fstream.h>
#include <time.h>
#include <ga/ga.h>

#include <mpi++.h>
int numprocs,myid,connexion;
MPI_Request request_list[100];
MPI_Status statu;

int MultiPar_EVALUATED_GENERATIONS=0;
int MultiPar_NB_EVALUATIONS=0;
clock_t   MultiPar_START, MultiPar_FINISH;
double MultiPar_LET=0; // Last Elapsed Time
double MultiPar_LERT=1; // Last Evaluated Remaining Time
float ReplPerc=0;
int NbGen, PSize;

// User Specific
// User Functions and Declarations

#include <vector>
int SIZE, phas_init, nopara, sizpop, noparaForm[4], optiform;
double fit;
vector<int>nbit(100);
vector<int>nbitForm(100);
vector<double>coef1(100);
vector<double>coef2(100);
vector<double>coef3(100);
vector<double>coefForm1(100);
```

```
vector<double>coefForm2(100);
vector<double>coefForm3(100);
extern "C" void objfun_();

// Initialisation function

void EASEAInitFunction(int argc, char *argv[]){
/*******************************************************************
Entree des donnees necessaires a l'algorithme genetique :
elles sont intouchables. Elles sont probleme-dependant.
-------------------------------------------------------
nopara = nombre de parametres,
coef1(i) = min(i) et coef2(i) = max(i) (i=1,nopara) des parametres,
coef3(i) =  inverse de la resolution (2**nbit(i) - 1)/(max(i)-min(i)),
nbit(i) = nombre de bits determinant la chaine binaire pour chaque parametre,
SIZE = taille des chromosomes
Ces donnees sont sauvegardees dans le fichier "info.dat"
*******************************************************************/
  double ddt;
  int i,j;
  std::ifstream entree("entree.dat0");  //Ouverture du fichier d'entree
  entree >> nopara;  //Lecture du nombre de parametres
  double* x=new double[nopara];
  double* precision=new double[nopara];
  for (i=0;i<nopara;i++)
  {
    entree >> coef1[i];  //Lecture des coef1(i)
  }
  for (i=0;i<nopara;i++)
  {
    entree >> coef2[i];  //Lecture des coef2(i)
  }
  for (i=0;i<nopara;i++)
  {
    entree >> precision[i];  //Lecture des precision(i)
  }
  std::ofstream info("info.dat");  //Ouverture du fichier "info.dat"
  info << nopara << endl;  //Sauvegarde du nombre de parametres
  for(i=0;i<nopara;i++)
  {
    ddt=fabs(coef2[i]-coef1[i])/precision[i];
    nbit[i]=int(log(ddt)/log(2.0))+1;
    SIZE+=nbit[i];
    coef3[i]=(pow(2,nbit[i])-1)/fabs(coef2[i]-coef1[i]);
    info << nbit[i] << endl;  //Sauvegarde des nbit(i)
    info << coef1[i] << endl;  //Sauvegarde des coef1(i)
    info << coef3[i] << endl;  //Sauvegarde des coef3(i)
  }
  delete[] precision;

  entree >> optiform;
  info << optiform << endl;;
/*******************************************************
Lecture des parametres de forme
Les parametres sont identiques pour les points de controle
d'une meme surface
*******************************************************/
  double* dy[4]; //dy[0]=dyBecSup; dy[1]=dyBecInf; dy[2]=dyVoletSup; dy[3]=dyVoletInf
  if (optiform==1)
  {
    double precisionForm;
    for (i=0;i<4;i++) //4 est le nombre de surfaces a parametriser
    {
```

```
      entree >> noparaForm[i];
      info << noparaForm[i] << endl;
      dy[i]=new double[noparaForm[i]];
      entree >> coefForm1[i];
      entree >> coefForm2[i];
      entree >> precisionForm;
      ddt=fabs(coefForm2[i]-coefForm1[i])/precisionForm;
      nbitForm[i]=int(log(ddt)/log(2.0))+1;
      coefForm3[i]=(pow(2,nbitForm[i])-1)/fabs(coefForm2[i]-coefForm1[i]);
      SIZE+=nbitForm[i];
      for (j=i+1;j<i+noparaForm[i];j++)
      {
SIZE+=nbitForm[i];
coefForm1[j]=coefForm1[i];
coefForm2[j]=coefForm2[i];
coefForm3[j]=coefForm3[i];
      }
      info << nbitForm[i] << endl;
      info << coefForm1[i] << endl;
      info << coefForm3[i] << endl;
      delete[] dy[i];
    }
  }
  objfun_(&phas_init,&nopara,&sizpop,x,&fit,&optiform,dy[0],dy[1],dy[2],dy[3]);  //Calcul de flot initial
  delete[] x;
  entree.close();
  info.close();
}

// User Classes


inline int random(int b1=0, int b2=1){
  return GARandomInt(b1,b2);
}
inline double random(double b1=0, double b2=1){
  return GARandomDouble(b1,b2);
}
inline float random(float b1=0, float b2=1){
  return GARandomFloat(b1,b2);
}

// User Genome
class MultiParGenome : public GAGenome {
// Default methods for class MultiParGenome
public:
  GADefineIdentity("MultiParGenome", 251);
  static void Initializer(GAGenome&);
  static int Mutator(GAGenome&, float);
  static float Comparator(const GAGenome&, const GAGenome&);
  static float Evaluator(GAGenome&);
  static int Crossover(const GAGenome&, const GAGenome&, GAGenome*, GAGenome*);
  void Decoder(char* fichier) const;
public:
  MultiParGenome::MultiParGenome(int l) :GAGenome(Initializer, Mutator, Comparator){
    evaluator(Evaluator); crossover(Crossover);
    len=l;
  }
  MultiParGenome(const MultiParGenome & orig) {
    copy(orig);
  }
  ~MultiParGenome() {
// Destructing pointers
```

```
  }
  MultiParGenome& operator=(const MultiParGenome &);
  virtual GAGenome *clone(GAGenome::CloneMethod) const ;
  virtual void copy(const GAGenome & c);
  virtual int equal(const GAGenome& g) const;
  virtual int read(istream & is);
  virtual int write(ostream & os) const ;

// Class members
  bool y[400];
  int len;
};

MultiParGenome& MultiParGenome::operator=(const MultiParGenome & arg){
  copy(arg);
  return *this;
}

void MultiParGenome::copy(const GAGenome& g) {
  if(&g != this){
    GAGenome::copy(g);    // copy the base class part
    MultiParGenome & genome = (MultiParGenome &)g;
    len=genome.len;
// Memberwise copy
    {for(int EASEA_Ndx=0; EASEA_Ndx<genome.len; EASEA_Ndx++)
       y[EASEA_Ndx]=genome.y[EASEA_Ndx];}
  }
}

GAGenome*MultiParGenome::clone(GAGenome::CloneMethod) const {
  return new MultiParGenome(*this);
}

int MultiParGenome::equal(const GAGenome& g) const {
  MultiParGenome& genome = (MultiParGenome&)g;
// Default diversity test (required by GALib)
  {for(int EASEA_Ndx=0; EASEA_Ndx<genome.len; EASEA_Ndx++)
     if (y[EASEA_Ndx]!=genome.y[EASEA_Ndx]) return 0;}
  return 1;
}

float MultiParGenome::Comparator(const GAGenome& a, const GAGenome& b) {
  MultiParGenome& sis = (MultiParGenome &)a;
  MultiParGenome& bro = (MultiParGenome &)b;
  int diff = 0;
// Default genome comparator (required by GALib)
  {for(int EASEA_Ndx=0; EASEA_Ndx<sis.len; EASEA_Ndx++)
     if (sis.y[EASEA_Ndx]!=bro.y[EASEA_Ndx]) diff++;}
  return (float)diff;
}

int MultiParGenome::read(istream & is) {
// Default read command (required by GALib)
  return is.fail() ? 1 : 0;
}

int MultiParGenome::write(ostream & os) const {
// Default write command (required by GALib)
  {os << "Array y : ";
   for(int EASEA_Ndx=0; EASEA_Ndx<len; EASEA_Ndx++)
     os << "[" << EASEA_Ndx << "]:" << y[EASEA_Ndx] << "\t";}
  os << "\n";
  return os.fail() ? 1 : 0;
```

```
}


// Standard Functions

void MultiParGenome::Initializer(GAGenome& g) {
  MultiParGenome & genome = (MultiParGenome &)g;
 // "initializer" is also accepted
  for (int i=0;i<genome.len;i++) genome.y[i]=GAFlipCoin(.5)?1:0;

genome._evaluated=gaFalse;
}

int MultiParGenome::Crossover(const GAGenome& a, const GAGenome& b, GAGenome* c, GAGenome* d) {
  MultiParGenome& mom = (MultiParGenome &)a;
  MultiParGenome& dad = (MultiParGenome &)b;
  MultiParGenome& sis = (MultiParGenome &)*c;
  MultiParGenome& bro = (MultiParGenome &)*d;
  if(&bro) bro._evaluated=gaFalse;
  if(&sis) sis._evaluated=gaFalse;
 // Must return the number of concerned children
  int GeneratedChildren=0;
  int CrossoverPosition=random(0,mom.len);
  if (&bro){
    for(int i=0;i<mom.len;i++)
      if (i<CrossoverPosition) bro.y[i]=dad.y[i];
      else bro.y[i]=mom.y[i];
    GeneratedChildren++;
  }
  if (&sis){
    for(int i=0;i<mom.len;i++)
      if (i<CrossoverPosition) sis.y[i]=mom.y[i];
      else sis.y[i]=dad.y[i];
    GeneratedChildren++;
  }
  return GeneratedChildren;

}

int MultiParGenome::Mutator(GAGenome& g, float pmut) {
  MultiParGenome & genome = (MultiParGenome &)g;
  genome._evaluated=gaFalse;
 // Must return the number of mutations
  int NbMut=0;
  for (int i=0;i<genome.len;i++)
    if (GAFlipCoin(pmut)){
       NbMut++;
       genome.y[i]=genome.y[i]?0:1;
    }
  return NbMut;
}

float MultiParGenome::Evaluator(GAGenome & c) {
/*************************************************************************
On decode une chaine binaire en la valeur reelle des parametres
x (reel) = xmin + (valeur de la chaine binaire)*(xmax-xmin)/(2**lchrom-1)
valeur de la chaine binaire = Somme (de i = 0 a lchrom-1) ai * 2**i
ou ai = 0 ou 1, ensuite on appelle le solveur qui renvoie la fitness
On lit dans "info.dat" les informations necessaires au decodage
*************************************************************************/
  int sbit,ebit,powerof2,i,j,k;
  double accum;
  sbit=0;
```

```
  ebit=0;
  MultiParGenome & genome = (MultiParGenome &)c;
  MultiPar_NB_EVALUATIONS++;
 // Must return (float) the score as a double
  std::ifstream info("info.dat");
  info >> nopara;
  double *x=new double[nopara];
  for (i=0;i<nopara;i++)
  {
    info >> nbit[i];
    info >> coef1[i];
    info >> coef3[i];
    ebit+=nbit[i];
    accum=0;
    powerof2=1;
    for (j=sbit;j<ebit;j++)
    {
      accum=accum+genome.y[j]*powerof2;
      powerof2=powerof2*2;
    }
    x[i]=accum/coef3[i];
    x[i]=x[i]+coef1[i];
    sbit=ebit;
  }

/*****************************
Decodage des parametres de forme
*****************************/
  double* dy[4];
  info >> optiform;
  if (optiform==1)
  {
    for (i=0;i<4;i++)
    {
      info >> noparaForm[i];
      info >> nbitForm[i];
      info >> coefForm1[i];
      info >> coefForm3[i];
      dy[i]=new double[noparaForm[i]];
      for (j=0;j<noparaForm[i];j++)
      {
ebit+=nbitForm[i];
accum=0;
powerof2=1;
for (k=sbit;k<ebit;k++)
{
  accum=accum+genome.y[k]*powerof2;
  powerof2=powerof2*2;
}
dy[i][j]=accum/coefForm3[i];
dy[i][j]+=coefForm1[i];
sbit=ebit;
      }
    }
  }
  info.close();
  phas_init=1;
  objfun_(&phas_init,&nopara,&sizpop,x,&fit,&optiform,dy[0],dy[1],dy[2],dy[3]);  //Appel du solveur
  delete[] x;
  if (optiform==1)
  {
    for (i=0;i<4;i++)
    {
```

```
      delete[] dy[i];
    }
  }
  return (float) fit;
}


void MultiParGenome::Decoder(char* fichier) const
{
/*****************************************************************************
On decode une chaine binaire en la valeur reelle des parametres
x (reel) = xmin + (valeur de la chaine binaire)*(xmax-xmin)/(2**lchrom-1)
valeur de la chaine binaire = Somme (de i = 0 a lchrom-1) ai * 2**i
ou ai = 0 ou 1, ensuite les x sont sauvegardes dans un fichier
On lit dans "info.dat" les informations necessaires au decodage
*****************************************************************************/
  int sbit,ebit,powerof2,i,j,k;
  double accum;
  sbit=0;
  ebit=0;
  std::ofstream parametres(fichier,ofstream::app);  //Ouverture du fichier et ajout en fin
  std::ifstream info("info.dat");
  info >> nopara;
  double* x=new double[nopara];
  for (i=0;i<nopara;i++)
  {
    info >> nbit[i];
    info >> coef1[i];
    info >> coef3[i];
    ebit+=nbit[i];
    accum=0;
    powerof2=1;
    for (j=sbit;j<ebit;j++)
    {
      accum=accum+y[j]*powerof2;
      powerof2=powerof2*2;
    }
    x[i]=accum/coef3[i];
    x[i]=x[i]+coef1[i];
    sbit=ebit;
    parametres << x[i] << " ";
  }
  delete[] x;
  parametres << endl;

/*****************************
Decodage des parametres de forme
*****************************/
  double* dy[4];
  info >> optiform;
  if (optiform==1)
  {
    for (i=0;i<4;i++)
    {
      info >> noparaForm[i];
      info >> nbitForm[i];
      info >> coefForm1[i];
      info >> coefForm3[i];
      dy[i]=new double[noparaForm[i]];
      for (j=0;j<noparaForm[i];j++)
      {
ebit+=nbitForm[i];
accum=0;
```

```
powerof2=1;
for (k=sbit;k<ebit;k++)
{
  accum=accum+y[k]*powerof2;
  powerof2=powerof2*2;
}
dy[i][j]=accum/coefForm3[i];
dy[i][j]+=coefForm1[i];
sbit=ebit;
parametres << dy[i][j] << " ";
      }
      delete[] dy[i];
      parametres << endl;
    }
  }
  parametres.close();
  info.close();
}



void trucEvaluator(GAPopulation & p){
  int i,index,k;
  int recept[numprocs];
  int nbr_procs_utils;
  float fit[numprocs];
  int qui_fait_quoi[numprocs];

  if (p.size()>numprocs-1)
    {nbr_procs_utils=numprocs-1;}
  else
    {nbr_procs_utils=p.size();}

  for(i=1;i<=nbr_procs_utils;i++) MPI_Send(&connexion,1,MPI_INT,i,100,MPI_COMM_WORLD);

  for(i=1;i<=nbr_procs_utils;i++){
    MPI_Send(&p.individual(i-1),sizeof(MultiParGenome),MPI_BYTE,i,200,MPI_COMM_WORLD);
    qui_fait_quoi[i]=i-1;
  }
  for(i=1;i<=nbr_procs_utils;i++) MPI_Irecv(&fit[i],1,MPI_FLOAT,i,300,MPI_COMM_WORLD,&request_list[i]);

  i=0;
  k=nbr_procs_utils;
  fflush(stdout);
  while(i<p.size())
    {
      MPI_Waitany(numprocs-1,&request_list[1],&index,&statu);
      fflush(stdout);
      p.individual(qui_fait_quoi[index+1]).score(fit[index+1]);

      if (k<p.size()){
MPI_Send(&connexion,1,MPI_INT,index+1,100,MPI_COMM_WORLD);
MPI_Send(&p.individual(k),sizeof(MultiParGenome),MPI_BYTE,index+1,200,MPI_COMM_WORLD);
MPI_Irecv(&fit[index+1],1,MPI_FLOAT,index+1,300,MPI_COMM_WORLD,&request_list[index+1]);
qui_fait_quoi[index+1]=k;
k++;
  }
      i++;
      fflush(stdout);
    }
}
```

```
int main(int argc, char *argv[]){
  FILE *MultiPar_PRM;
  int i;

  //     MPI**********************
  float fit;
  MPI_Init( &argc , &argv );
  MPI_Comm_rank( MPI_COMM_WORLD , &myid);
  MPI_Comm_size( MPI_COMM_WORLD , &numprocs );

  GAGenome gen;
  if (myid==0)
    {
      connexion=1;
  //     MPI**********************

  GARandomSeed(0);
  EASEAInitFunction(argc, argv);
  MultiParGenome genome(SIZE);

// Checks whether we've been given a seed to use (for testing purposes).
  for(i=1; i<argc; i++)
    if(strcmp(argv[i++],"seed") == 0)
      GARandomSeed((unsigned int)atoi(argv[i]));
  GAPopulation populatio(genome,30);
  populatio.evaluator(trucEvaluator);

  GASteadyStateGA ga(populatio);

  GASRSSelector SRSSelector;  //echantillonnage stochastique sans remplacement
  GATournamentSelector TournamentSelector;  //Selection par tournoi
  GADSSelector DSSelector;  //echantillonnage deterministe
  ga.selector(SRSSelector); //specifier ici la methode de selection (defaut:roulette)

  GAPowerLawScaling PowerLawScaling();  //specifier le coefficient (defaut=1.0005)
  GALinearScaling LinearScaling();  //specifier le multiplicateur (defaut=1.2)
  GASharing Sharing;
  Sharing.sigma();  //Specifier le rayon de la niche
  Sharing.alpha();  //Specifier le degre de la fonction de partage (defaut=1)
  ga.scaling();  //specifier la methode de modification de la fitness pour AG1

  ga.scoreFilename("maxavgmin.dat");
  ga.selectScores(GAStatistics::AllScores);
  ga.scoreFrequency(1);
  ga.flushFrequency(1);
  ga.populationSize(30); // how many individuals in the population
  ga.nGenerations(40);    // number of generations to evolve
  ga.pReplacement((float)90.000000/100);    // percentage of the population to be replaced
  ga.pMutation((float)0.0500000);    // likelihood of mutating new offspring
  ga.pCrossover((float)0.900000);   // likelihood of crossing over parents
  if (MultiPar_PRM=fopen("MultiPar.prm","r")){
    fclose(MultiPar_PRM);
    ga.parameters("MultiPar.prm"); // gets GALib parameters from the MultiPar.prm file
  }
  ga.parameters(argc, argv); // gets GALib parameters from the command line
  PSize=ga.populationSize();
  NbGen=ga.nGenerations();
  ReplPerc=ga.pReplacement()*100;

  genome.initialize();
  cout << "Score of a generation 0 genome: " << genome.evaluate(gaTrue) << "\n";
  cout << "Contents of the genome:\n" << genome << endl;
  MultiPar_START=clock();
```

```
  ga.initialize();
  do{
    ga.step();
    MultiPar_EVALUATED_GENERATIONS++;
    MultiPar_FINISH=clock();
    if ((double)(MultiPar_FINISH-MultiPar_START)-MultiPar_LET > CLOCKS_PER_SEC){
      for (i=0;i<log10(MultiPar_LERT);i++) fprintf(stderr,"\b");
      MultiPar_LET=(double)(MultiPar_FINISH-MultiPar_START) ;
      MultiPar_LERT=(MultiPar_LET*(ga.nGenerations()-MultiPar_EVALUATED_GENERATIONS)/MultiPar_EVALUATED_GENERATIONS);
      MultiPar_LERT=/CLOCKS_PER_SEC;
    }
  } while(!ga.done());
  cout << "\nBest genome score: " << (ga.statistics().bestIndividual()).evaluate() << endl;
  cout << "Contents of the genome:\n" << ga.statistics().bestIndividual();
  cout << "Nombre d'evaluations : " << MultiPar_NB_EVALUATIONS << endl;
  cout << "Main parameters: PSize=" << PSize << ", NbGen=" << NbGen <<", PMut=" << ga.pMutation();
  cout << "PCross=" << ga.pCrossover();
  cout << ", ReplPerc=" << ReplPerc;
  cout << "\n";
/************************Calcul optimal*********************************************/
  sizpop=1;
  (ga.statistics().bestIndividual()).evaluate(gaTrue);
/************************************************************************************/
      connexion=0;
      for(i=1;i<=numprocs-1;i++) MPI_Send(&connexion,1,MPI_INT,i,100,MPI_COMM_WORLD);
    }
  else
    {
      connexion=1;
      while (connexion!=0){
MPI_Recv(&connexion,1,MPI_INT,0,100,MPI_COMM_WORLD,&statu);
if (connexion==1){
  MPI_Recv(&gen,sizeof(MultiParGenome),MPI_BYTE,0,200,MPI_COMM_WORLD,&statu);
  fit=gen.score();
  MPI_Send(&fit,1,MPI_FLOAT,0,300,MPI_COMM_WORLD);}
      }
   }

 //     MPI*********************
 MPI_Finalize();
 //     MPI*********************
  exit(0);
  return 0;
}

// If your compiler does not do automatic instantiation (e.g. g++ 2.6.8),
// then define the NO_AUTO_INST directive.  This will force the instantiation
// of the template classes that we use.  For some compilers (e.g. metrowerks)
// this must come after any specializations or you'll get 'multiply-defined'
// errors when you compile.
#ifdef NO_AUTO_INST
#include "GAList.cpp"
#include "GAListGe.cpp"
#if defined(__GNUG__)
template class GAList<int>;
template class GAListGenome<int>;
#else
GAList<int>;
GAListGenome<int>;
#endif
#endif
```

```
// That's all folks !
```

Annexe D

Code d'optimisation parallèle, stratégie de Nash

```
//////////////////////////////////////////
//
//  MultiParNash.cpp
//
//  C++ file generated by EASEA-GALIB v0.4
//
//////////////////////////////////////////

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <iostream.h>
#include <fstream.h>
#include <time.h>
#include <ga/ga.h>

#include <mpi++.h>
int numprocs,myid,connexion;
MPI_Request request_list[100];
MPI_Status statu;

int MultiParNash_EVALUATED_GENERATIONS=0;
int MultiParNash_NB_EVALUATIONS=0;
clock_t   MultiParNash_START, MultiParNash_FINISH;
double MultiParNash_LET=0; // Last Elapsed Time
double MultiParNash_LERT=1; // Last Evaluated Remaining Time
float ReplPerc=0;
int NbGen, PSize;

// User Specific
// User Functions and Declarations

#include <vector>
  int SIZE1,SIZE2,phas_init, nopara, sizpop;
  double fit;
  vector<int>nbit(100);
```

```
  vector<double>coef1(100);
  vector<double>coef2(100);
  vector<double>coef3(100);
  extern "C" void objfun_();

// Initialisation function

void EASEAInitFunction(int argc, char *argv[]){
/*******************************************************************
Entree des donnees necessaires aux algorithmes genetiques :
elles sont intouchables. Elles sont probleme-dependant.
---------------------------------------------------
nopara = nombre de parametres,
coef1(i) = min(i) et coef2(i) = max(i) (i=1,nopara) des parametres,
coef3(i) =  inverse de la resolution (2**nbit(i) - 1)/(max(i)-min(i)),
nbit(i) = nombre de bits determinant la chaine binaire pour chaque parametre,
SIZE1 = taille des chromosomes pour l'AG 1
SIZE2 = taille des chromosomes pour l'AG 2
Ces donnees sont sauvegardees dans le fichier "info.dat"
"initilisation.dat" contient une solution initiale
*******************************************************************/
/*On lit d'abord les parametres pour l'AG 1 qui optimise la position
du volet, la position du bec etant fixe*/
  int i,k;
  double ddt;
  std::ifstream entree1("entree1.dat");  //Ouverture du fichier d'entree de l'AG 1
  entree1 >> nopara;  //Lecture du nombre de parametres de l'AG 1
  double* x=new double[2*nopara];
  double* precision=new double[nopara];
  int eval=1; //eval=1 evaluation pour l'AG 1
              //eval=2 evaluation pour l'AG 2
  nopara=2*nopara;
  objfun_(&phas_init,&nopara,&sizpop,x,&fit,&eval);  //Calcul de flot initial
  nopara=nopara/2;
  for (i=0;i<nopara;i++)
  {
    entree1 >> coef1[i];  //Lecture des coef1(i)
  }
  for (i=0;i<nopara;i++)
  {
    entree1 >> coef2[i];  //Lecture des coef2(i)
  }
  for (i=0;i<nopara;i++)
  {
    entree1 >> precision[i];  //Lecture des precision(i)
  }
  entree1.close();
  std::ofstream info1("info1.dat");  //Ouverture du fichier "info-att.dat"
                                     //ou sont sauvegardes les parametres de l'AG1
  info1 << nopara << endl;  //Sauvegarde du nombre de parametres
  for(i=0;i<nopara;i++)
  {
    ddt=fabs(coef2[i]-coef1[i])/precision[i];
    nbit[i]=int(log(ddt)/log(2.0))+1;
    SIZE1=SIZE1+nbit[i];
    coef3[i]=(pow(2,nbit[i])-1)/fabs(coef2[i]-coef1[i]);
    info1 << nbit[i] << endl;  //Sauvegarde des nbit(i)
    info1 << coef1[i] << endl;  //Sauvegarde des coef1(i)
    info1 << coef3[i] << endl;  //Sauvegarde des coef3(i)
  }
  info1.close();

  std::ofstream best1("best1.dat"); //Ouverture du fichier "best1.dat" ou est
```

```
                                    //sauvegarde le meilleur genome trouve par l'AG 1
                                    //il est initialise de maniere aleatoire
  for (i=0;i<SIZE1;i++)
  {
    best1 << GAFlipCoin(.5)?1:0;
    best1 << " ";
  }
  best1.close();

/*On lit ensuite les parametres pour l'AG 2 qui optimise la position
du bec, la position du volet etant fixe*/
  std::ifstream entree2("entree2.dat");  //Ouverture du fichier d'entree de l'AG 2
  entree2 >> nopara;  //Lecture du nombre de parametres de l'AG 2
  eval=2;
  phas_init=0;
  nopara=2*nopara;
  objfun_(&phas_init,&nopara,&sizpop,x,&fit,&eval);
  nopara=nopara/2;
  for (i=0;i<nopara;i++)
  {
    entree2 >> coef1[i];
  }
  for (i=0;i<nopara;i++)
  {
    entree2 >> coef2[i];
  }
  for (i=0;i<nopara;i++)
  {
    entree2 >> precision[i];
  }
  entree2.close();
  std::ofstream info2("info2.dat");//Ouverture du fichier "info-dec.dat"
                                    //ou sont sauvegardes les parametres de
                                    //l'AG 2
  info2 << nopara << endl;
  for(i=0;i<nopara;i++)
  {
    ddt=fabs(coef2[i]-coef1[i])/precision[i];
    nbit[i]=int(log(ddt)/log(2.0))+1;
    SIZE2=SIZE2+nbit[i];
    bool* temp=new bool[nbit[i]];
    coef3[i]=(pow(2,nbit[i])-1)/fabs(coef2[i]-coef1[i]);
    info2 << nbit[i] << endl;
    info2 << coef1[i] << endl;
    info2 << coef3[i] << endl;
  }
  info2.close();

  std::ofstream best2("best2.dat"); //Ouverture du fichier "best2.dat" ou est
                                    //sauvegarde le meilleur genome trouve par l'AG 2
                                    //il est initialise de maniere aleatoire
  for (i=0;i<SIZE2;i++)
  {
    best2 << GAFlipCoin(.5)?1:0;
    best2 << " ";
  }
  best2.close();
  delete[] x;
  delete[] precision;
}

// User Classes
```

```
inline int random(int b1=0, int b2=1){
  return GARandomInt(b1,b2);
}
inline double random(double b1=0, double b2=1){
  return GARandomDouble(b1,b2);
}
inline float random(float b1=0, float b2=1){
  return GARandomFloat(b1,b2);
}

// User Genome
class MultiParNashGenome : public GAGenome {
// Default methods for class MultiParNashGenome
public:
  GADefineIdentity("MultiParNashGenome", 251);
  static void Initializer(GAGenome&);
  static int Mutator(GAGenome&, float);
  static float Comparator(const GAGenome&, const GAGenome&);
  static float Evaluator(GAGenome&);
  static int Crossover(const GAGenome&, const GAGenome&, GAGenome*, GAGenome*);
  void Decoder(char* fichier) const;
public:
  MultiParNashGenome::MultiParNashGenome(int l, int f) :GAGenome(Initializer, Mutator, Comparator){
    evaluator(Evaluator); crossover(Crossover);
    len=l;
    evalf=f;
  }
  MultiParNashGenome(const MultiParNashGenome & orig) {
    copy(orig);
  }
  ~MultiParNashGenome() {
// Destructing pointers
  }
  MultiParNashGenome& operator=(const MultiParNashGenome &);
  virtual GAGenome *clone(GAGenome::CloneMethod) const ;
  virtual void copy(const GAGenome & c);
  virtual int equal(const GAGenome& g) const;
  virtual int read(istream & is);
  virtual int write(ostream & os) const ;

// Class members
  bool y[100];
  int len;
  int evalf;
};

MultiParNashGenome& MultiParNashGenome::operator=(const MultiParNashGenome & arg){
  copy(arg);
  return *this;
}

void MultiParNashGenome::copy(const GAGenome& g) {
  if(&g != this){
    GAGenome::copy(g);    // copy the base class part
    MultiParNashGenome & genome = (MultiParNashGenome &)g;
    len=genome.len;
    evalf=genome.evalf;
// Memberwise copy
    {for(int EASEA_Ndx=0; EASEA_Ndx<genome.len; EASEA_Ndx++)
       y[EASEA_Ndx]=genome.y[EASEA_Ndx];}
  }
}
```

```
GAGenome*MultiParNashGenome::clone(GAGenome::CloneMethod) const {
  return new MultiParNashGenome(*this);
}

int MultiParNashGenome::equal(const GAGenome& g) const {
  MultiParNashGenome& genome = (MultiParNashGenome&)g;
// Default diversity test (required by GALib)
  {for(int EASEA_Ndx=0; EASEA_Ndx<genome.len; EASEA_Ndx++)
     if (y[EASEA_Ndx]!=genome.y[EASEA_Ndx]) return 0;}
  return 1;
}

float MultiParNashGenome::Comparator(const GAGenome& a, const GAGenome& b) {
  MultiParNashGenome& sis = (MultiParNashGenome &)a;
  MultiParNashGenome& bro = (MultiParNashGenome &)b;
  int diff = 0;
// Default genome comparator (required by GALib)
  {for(int EASEA_Ndx=0; EASEA_Ndx<sis.len; EASEA_Ndx++)
     if (sis.y[EASEA_Ndx]!=bro.y[EASEA_Ndx]) diff++;}
  return (float)diff;
}

int MultiParNashGenome::read(istream & is) {
// Default read command (required by GALib)
  return is.fail() ? 1 : 0;
}

int MultiParNashGenome::write(ostream & os) const {
// Default write command (required by GALib)
  {os << "Array y : ";
   for(int EASEA_Ndx=0; EASEA_Ndx<len; EASEA_Ndx++)
     os << "[" << EASEA_Ndx << "]:" << y[EASEA_Ndx] << "\t";}
  os << "\n";
  return os.fail() ? 1 : 0;
}


// Standard Functions

void MultiParNashGenome::Initializer(GAGenome& g) {
  MultiParNashGenome & genome = (MultiParNashGenome &)g;
 // "initializer" is also accepted
  for (int i=0;i<genome.len;i++) genome.y[i]=GAFlipCoin(.5)?1:0;

genome._evaluated=gaFalse;
}

int MultiParNashGenome::Crossover(const GAGenome& a, const GAGenome& b, GAGenome* c, GAGenome* d) {
  MultiParNashGenome& mom = (MultiParNashGenome &)a;
  MultiParNashGenome& dad = (MultiParNashGenome &)b;
  MultiParNashGenome& sis = (MultiParNashGenome &)*c;
  MultiParNashGenome& bro = (MultiParNashGenome &)*d;
  if(&bro) bro._evaluated=gaFalse;
  if(&sis) sis._evaluated=gaFalse;
 // Must return the number of concerned children
  int GeneratedChildren=0;
  int CrossoverPosition=random(0,mom.len);
  if (&bro){
    for(int i=0;i<mom.len;i++)
      if (i<CrossoverPosition) bro.y[i]=dad.y[i];
      else bro.y[i]=mom.y[i];
    GeneratedChildren++;
```

```
  }
  if (&sis){
    for(int i=0;i<mom.len;i++)
      if (i<CrossoverPosition) sis.y[i]=mom.y[i];
      else sis.y[i]=dad.y[i];
    GeneratedChildren++;
  }
  return GeneratedChildren;

}

int MultiParNashGenome::Mutator(GAGenome& g, float pmut) {
  MultiParNashGenome & genome = (MultiParNashGenome &)g;
  genome._evaluated=gaFalse;
 // Must return the number of mutations
  int NbMut=0;
  for (int i=0;i<genome.len;i++)
    if (GAFlipCoin(pmut)){
       NbMut++;
       genome.y[i]=genome.y[i]?0:1;
    }
  return NbMut;
}

float MultiParNashGenome::Evaluator(GAGenome & c) {
/*************************************************************************
On decode une chaine binaire en la valeur reelle des parametres
x (reel) = xmin + (valeur de la chaine binaire)*(xmax-xmin)/(2**lchrom-1)
valeur de la chaine binaire = Somme (de i = 0 a lchrom-1) ai * 2**i
ou ai = 0 ou 1, ensuite on appelle le solveur qui renvoie la fitness
On lit dans "info-att.dat" les informations necessaires au decodage pour l'AG 1
et dans "info-dec.dat" pour l'AG 2
Le genome de l'AG 1 est complete avec le fichier "best2.dat" (meilleur genome de l'AG 2)
Le genome de l'AG 2 est complete avec le fichier "best1.dat" (meilleur genome de l'AG 1)
*************************************************************************/
  int i,j,sbit,ebit,powerof2,var;
  double accum;
  sbit=0;
  ebit=0;
  MultiParNashGenome & genome = (MultiParNashGenome &)c;
  MultiParNash_NB_EVALUATIONS++;
 // Must return (float) the score as a double
  char* ficbest;
  char* ficinfo1;
  char* ficinfo2;
  if (genome.evalf==1)
  {
    ficbest="best2.dat";
    ficinfo1="info1.dat";
    ficinfo2="info2.dat";
  }
  if (genome.evalf==2)
  {
    ficbest="best1.dat";
    ficinfo1="info2.dat";
    ficinfo2="info1.dat";
  }
  std::ifstream best(ficbest);
  std::ifstream info1(ficinfo1);
  std::ifstream info2(ficinfo2);
  info1 >> nopara;
  info2 >> nopara;
  double *x=new double[2*nopara];
```

```
  for (i=0;i<2*nopara;i++)
  {
    if (i<nopara)
    {
      info1 >> nbit[i];
      info1 >> coef1[i];
      info1 >> coef3[i];
      ebit+=nbit[i];
      accum=0;
      powerof2=1;
      for (j=sbit;j<ebit;j++)
      {
accum=accum+genome.y[j]*powerof2;
powerof2=powerof2*2;
      }
      x[i]=accum/coef3[i];
      x[i]=x[i]+coef1[i];
      sbit=ebit;
    }
    else
    {
      info2 >> nbit[i];
      info2 >> coef1[i];
      info2 >> coef3[i];
      ebit+=nbit[i];
      accum=0;
      powerof2=1;
      for (j=sbit;j<ebit;j++)
      {
best >> var;
accum=accum+var*powerof2;
powerof2=powerof2*2;
      }
      x[i]=accum/coef3[i];
      x[i]=x[i]+coef1[i];
      sbit=ebit;
    }
  }
  best.close();
  info1.close();
  info2.close();
  if (genome.evalf==1)
  {
    double tmp;
    for (i=0;i<nopara;i++)
    {
      tmp=x[i];
      x[i]=x[i+nopara];
      x[i+nopara]=tmp;
    }
  }

  phas_init=1;
  nopara=2*nopara;
  objfun_(&phas_init,&nopara,&sizpop,x,&fit,&genome.evalf);
  nopara=nopara/2;
  delete[] x;
  return (float) fit;
}


void MultiParNashGenome::Decoder(char* fichier) const
/*****************************************************************************
```

```
On decode une chaine binaire en la valeur reelle des parametres
x (reel) = xmin + (valeur de la chaine binaire)*(xmax-xmin)/(2**lchrom-1)
valeur de la chaine binaire = Somme (de i = 0 a lchrom-1) ai * 2**i
ou ai = 0 ou 1, ensuite les x sont sauvegardes dans un fichier
On lit dans "info-att.dat" les informations necessaires au decodage pour l'AG 1
et dans "info-dec.dat" pour l'AG 2
Le genome de l'AG 1 est complete avec le fichier "best2.dat" (meilleur genome de l'AG 2)
Le genome de l'AG 2 est complete avec le fichier "best1.dat" (meilleur genome de l'AG 1)
*****************************************************************************/
{
  char* ficbest;
  char* ficinfo1;
  char* ficinfo2;
  if (evalf==1)
  {
    ficbest="best2.dat";
    ficinfo1="info1.dat";
    ficinfo2="info2.dat";
  }
  if (evalf==2)
  {
    ficbest="best1.dat";
    ficinfo1="info2.dat";
    ficinfo2="info1.dat";
  }
  std::ifstream best(ficbest);
  std::ifstream info1(ficinfo1);
  std::ifstream info2(ficinfo2);
  info1 >> nopara;
  info2 >> nopara;
  int i,j;
  double *x=new double[2*nopara];
  int sbit,ebit,powerof2,var;
  double accum;
  sbit=0;
  ebit=0;
  for (i=0;i<2*nopara;i++)
  {
    if (i<nopara)
    {
      info1 >> nbit[i];
      info1 >> coef1[i];
      info1 >> coef3[i];
      ebit+=nbit[i];
      accum=0;
      powerof2=1;
      for (j=sbit;j<ebit;j++)
      {
accum=accum+y[j]*powerof2;
powerof2=powerof2*2;
      }
      x[i]=accum/coef3[i];
      x[i]=x[i]+coef1[i];
      sbit=ebit;
    }
    else
    {
      info2 >> nbit[i];
      info2 >> coef1[i];
      info2 >> coef3[i];
      accum=0;
      powerof2=1;
      for (j=0;j<nbit[i];j++)
```

```
      {
best >> var;
accum=accum+var*powerof2;
powerof2=powerof2*2;
      }
      x[i]=accum/coef3[i];
      x[i]=x[i]+coef1[i];
    }
  }
  info1.close();
  info2.close();
  best.close();
  if (evalf==1)
  {
    double tmp;
    for (i=0;i<nopara;i++)
    {
      tmp=x[i];
      x[i]=x[i+nopara];
      x[i+nopara]=tmp;
    }
  }
  std::ofstream parametres(fichier,ofstream::app);
  for (i=0;i<2*nopara;i++)
  {
    parametres << x[i] << " ";
  }
  parametres << endl;
  parametres.close();
}



void trucEvaluator(GAPopulation & p){
  int i,index,k;
  int recept[numprocs];
  int nbr_procs_utils;
  float fit[numprocs];
  int qui_fait_quoi[numprocs];

  if (p.size()>numprocs-1)
    {nbr_procs_utils=numprocs-1;}
  else
    {nbr_procs_utils=p.size();}

  for(i=1;i<=nbr_procs_utils;i++) MPI_Send(&connexion,1,MPI_INT,i,100,MPI_COMM_WORLD);

  for(i=1;i<=nbr_procs_utils;i++){
    MPI_Send(&p.individual(i-1),sizeof(MultiParNashGenome),MPI_BYTE,i,200,MPI_COMM_WORLD);
    qui_fait_quoi[i]=i-1;
  }
  for(i=1;i<=nbr_procs_utils;i++) MPI_Irecv(&fit[i],1,MPI_FLOAT,i,300,MPI_COMM_WORLD,&request_list[i]);

  i=0;
  k=nbr_procs_utils;
  fflush(stdout);
  while(i<p.size())
    {
      MPI_Waitany(numprocs-1,&request_list[1],&index,&statu);
      fflush(stdout);
      p.individual(qui_fait_quoi[index+1]).score(fit[index+1]);

      if (k<p.size()){
```

```
MPI_Send(&connexion,1,MPI_INT,index+1,100,MPI_COMM_WORLD);
MPI_Send(&p.individual(k),sizeof(MultiParNashGenome),MPI_BYTE,index+1,200,MPI_COMM_WORLD);
MPI_Irecv(&fit[index+1],1,MPI_FLOAT,index+1,300,MPI_COMM_WORLD,&request_list[index+1]);
qui_fait_quoi[index+1]=k;
k++;
  }
      i++;
      fflush(stdout);
    }
}


int main(int argc, char *argv[]){
  FILE *MultiParNash_PRM;
  int i,j;

  //     MPI**********************
  float fit;
  MPI_Init( &argc , &argv );
  MPI_Comm_rank( MPI_COMM_WORLD , &myid);
  MPI_Comm_size( MPI_COMM_WORLD , &numprocs );

  GAGenome genome;
  if (myid==0)
    {
      connexion=1;
  //     MPI**********************

  GARandomSeed(0);
  EASEAInitFunction(argc, argv);
  MultiParNashGenome genome1(SIZE1,1);
  MultiParNashGenome genome2(SIZE2,2);

// Checks whether we've been given a seed to use (for testing purposes).
  for(i=1; i<argc; i++)
    if(strcmp(argv[i++],"seed") == 0)
      GARandomSeed((unsigned int)atoi(argv[i]));

  GAPopulation populatio1(genome1,30);
  GAPopulation populatio2(genome2,30);
  populatio1.evaluator(trucEvaluator);
  populatio2.evaluator(trucEvaluator);

  GASteadyStateGA ga1(populatio1);
  GASteadyStateGA ga2(populatio2);

  GASRSSelector SRSSelector;  //echantillonnage stochastique sans remplacement
  GATournamentSelector TournamentSelector;  //Selection par tournoi
  GADSSelector DSSelector;  //echantillonnage deterministe
  ga1.selector(SRSSelector); //specifier ici la methode de selection pour AG1 (defaut:roulette)
  ga2.selector(SRSSelector); //specifier ici la methode de selection pour AG2 (defaut:roulette)

  GAPowerLawScaling PowerLawScaling(1.0005);  //specifier le coefficient (defaut=1.0005)
  GALinearScaling LinearScaling(1.2);  //specifier le multiplicateur (defaut=1.2)
  GASharing Sharing;
  Sharing.sigma();  //Specifier le rayon de la niche
  Sharing.alpha();  //Specifier le degre de la fonction de partage (defaut=1)
  ga1.scaling(PowerLawScaling);  //specifier la methode de modification de la fitness pour AG1
  ga2.scaling(PowerLawScaling);  //specifier la methode de modification de la fitness pour AG2

  ga1.scoreFilename("maxavgmin1.dat");
  ga1.selectScores(GAStatistics::AllScores);
  ga1.scoreFrequency(1);
```

```
ga1.flushFrequency(1);
ga1.populationSize(30); // how many individuals in the population
ga1.nGenerations(40);    // number of generations to evolve
ga1.pReplacement((float)90.000000/100);    // percentage of the population to be replaced
ga1.pMutation((float)0.050000);    // likelihood of mutating new offspring
ga1.pCrossover((float)0.900000);   // likelihood of crossing over parents

ga2.scoreFilename("maxavgmin2.dat");
ga2.selectScores(GAStatistics::AllScores);
ga2.scoreFrequency(1);
ga2.flushFrequency(1);
ga2.populationSize(30); // how many individuals in the population
ga2.nGenerations(40);    // number of generations to evolve
ga2.pReplacement((float)90.000000/100);    // percentage of the population to be replaced
ga2.pMutation((float)0.050000);    // likelihood of mutating new offspring
ga2.pCrossover((float)0.900000);   // likelihood of crossing over parents
if (MultiParNash_PRM=fopen("MultiParNash.prm","r")){
  fclose(MultiParNash_PRM);
  ga1.parameters("MultiParNash.prm"); // gets GALib parameters from the MultiParNash.prm file
  ga2.parameters("MultiParNash.prm");
}
ga1.parameters(argc, argv); // gets GALib parameters from the command line
ga2.parameters(argc, argv);
PSize=ga1.populationSize();
NbGen=ga1.nGenerations();
ReplPerc=ga1.pReplacement()*100;

genome1.initialize();
genome2.initialize();
cout << "Score of a generation 0 genome1: " << genome1.score() << "\n";
cout << "Contents of the genome1:\n" << genome1 << endl;
cout << "Score of a generation 0 genome2: " << genome2.score() << "\n";
cout << "Contents of the genome2:\n" << genome2 << endl;

MultiParNash_START=clock();
ga1.initialize();
ga2.initialize();
std::ofstream best1("best1.dat");
/*Sauvegarde du meilleur genome de l'AG 1*/
for (int j=0;j<((MultiParNashGenome &)(ga1.statistics().bestIndividual())).len;j++)
{
  best1 << ((MultiParNashGenome &)(ga1.statistics().bestIndividual())).y[j] << " ";
}
best1.close();

std::ofstream best2("best2.dat");
/*Sauvegarde du meilleur genome de l'AG 2*/
for (int j=0;j<((MultiParNashGenome&)(ga2.statistics().bestIndividual())).len;j++)
{
  best2 << ((MultiParNashGenome&)(ga2.statistics().bestIndividual())).y[j] << " ";
}
best2.close();

do{
  ga1.step();
  ga2.step();
  MultiParNash_EVALUATED_GENERATIONS++;
  MultiParNash_FINISH=clock();
  if ((double)(MultiParNash_FINISH-MultiParNash_START)-MultiParNash_LET > CLOCKS_PER_SEC){
    for (i=0;i<log10(MultiParNash_LERT);i++) fprintf(stderr,"\b");
    MultiParNash_LET=(double)(MultiParNash_FINISH-MultiParNash_START) ;
    MultiParNash_LERT=(MultiParNash_LET*(ga1.nGenerations()-MultiParNash_EVALUATED_GENERATIONS);
    MultiParNash_LERT=/MultiParNash_EVALUATED_GENERATIONS)/CLOCKS_PER_SEC;
```

```
    }
    std::ofstream best1("best1.dat");
    /*Sauvegarde du meilleur genome de l'AG 1*/
    for (int j=0;j<((MultiParNashGenome &)(ga1.statistics().bestIndividual())).len;j++)
    {
      best1 << ((MultiParNashGenome &)(ga1.statistics().bestIndividual())).y[j] << " ";
    }
    best1.close();

    std::ofstream best2("best2.dat");
    /*Sauvegarde du meilleur genome de l'AG 2*/
    for (int j=0;j<((MultiParNashGenome&)(ga2.statistics().bestIndividual())).len;j++)
    {
      best2 << ((MultiParNashGenome&)(ga2.statistics().bestIndividual())).y[j] << " ";
    }
    best2.close();
  } while((!ga1.done())&&(!ga2.done()));
  cout << "\nBest genome1 score: " << (ga1.statistics().bestIndividual()).score() << endl;
  cout << "Contents of the genome1:\n" << ga1.statistics().bestIndividual();
  cout << "\nBest genome2 score: " << (ga2.statistics().bestIndividual()).score() << endl;
  cout << "Contents of the genome2:\n" << ga2.statistics().bestIndividual();
  cout << "Number of evaluations :" << MultiParNash_NB_EVALUATIONS << endl;
/************************Calcul optimal*********************************************/
  sizpop=1;
  (ga1.statistics().bestIndividual()).evaluate(gaTrue);
  (ga2.statistics().bestIndividual()).evaluate(gaTrue);
/***********************************************************************************/
      connexion=0;
      for(i=1;i<=numprocs-1;i++) MPI_Send(&connexion,1,MPI_INT,i,100,MPI_COMM_WORLD);
    }
  else
    {
      connexion=1;
      while (connexion!=0){
MPI_Recv(&connexion,1,MPI_INT,0,100,MPI_COMM_WORLD,&statu);
if (connexion==1){
  MPI_Recv(&genome,sizeof(MultiParNashGenome),MPI_BYTE,0,200,MPI_COMM_WORLD,&statu);
  fit=genome.score();
  MPI_Send(&fit,1,MPI_FLOAT,0,300,MPI_COMM_WORLD);}
      }
   }

 //     MPI*********************
 MPI_Finalize();
 //     MPI*********************
  exit(0);
  return 0;
}

// If your compiler does not do automatic instantiation (e.g. g++ 2.6.8),
// then define the NO_AUTO_INST directive.  This will force the instantiation
// of the template classes that we use.  For some compilers (e.g. metrowerks)
// this must come after any specializations or you'll get 'multiply-defined'
// errors when you compile.
#ifdef NO_AUTO_INST
#include "GAList.cpp"
#include "GAListGe.cpp"
#if defined(__GNUG__)
template class GAList<int>;
template class GAListGenome<int>;
#else
GAList<int>;
GAListGenome<int>;
```

```
#endif
#endif


// That's all folks !
```

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - B.P. 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Lorraine : Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - B.P. 101 - 54602 Villers lès Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot St Martin (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, B.P. 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399